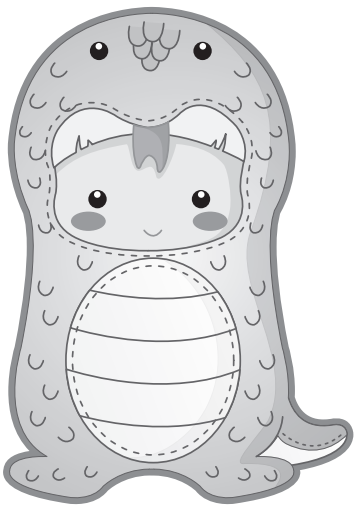
Python 3 学习笔记

上卷

雨痕 / 著



電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

VISIT…

内 容 简 介

经过 9 年的发展，Python 3 生态已相当成熟。无论是语言进化、解释器性能提升，还是第三方支持，都是如此。随着 Python 2.7 EOF 日趋临近，迁移到 Python 3 的各种障碍也被逐一剔除。是时候在新环境下学习或工作了。

人们常说 Python 简单易学，但这是以封装和隐藏复杂体系为代价的。仅阅读语言规范很难深入，亦无从发挥其应有能力，易学难精才是常态。本书尝试通过分析解释器的工作机制来解析 Python 3.6 语言理论，以期帮助读者加深理解。

本书着重于剖析语言的相关背景和实现方式，适合有一定 Python 编程基础的读者阅读、参考。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

Python 3 学习笔记. 上卷 / 雨痕著. —北京：电子工业出版社，2018.1
ISBN 978-7-121-33274-6

Ⅰ. ①P… Ⅱ. ①雨… Ⅲ. ①软件工具－程序设计 Ⅳ. ①TP311.561

中国版本图书馆 CIP 数据核字(2017)第 308906 号

策划编辑：许　艳
责任编辑：李云静
印　　刷：北京市京科印刷有限公司
装　　订：北京市京科印刷有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：787×980　1/16　印张：26.75　字数：500 千字
版　　次：2018 年 1 月第 1 版
印　　次：2018 年 1 月第 1 次印刷
定　　价：89.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

前言

写作本书时，我已然摆脱“萌新”身份，算是稍有经验的作者。可即便如此，我依然无法保证本书的内容完全正确，且满足所有人的胃口。显然，这不可能做到。

在我看来，图书大抵分两类：学习和研究。学习类书籍满足日常学习和提升需要，用简练的语言把问题说清楚。最关键的是有清晰的线索，把散乱的知识串联起来，学习者可据此了解前因后果。至于研究类图书或论文，则应摆脱基础，摆脱语法，重点关注算法、架构、性能，乃至内部实现。所有这些，均以思想为支撑，超脱语言窠臼，构建并完善体系。

不同于写散文或小说，技术类图书的文字不好组织。自然语言易阅读，但不便描述有复杂流程分支的逻辑，易导致歧义。更何况，这其中还有各种转译带来的麻烦。故技术类图书应以自然语言开宗明义，阐述理论与规则，随后用代码对这段文字进行解释，毕竟代码先天有描述逻辑的优势。

很多书，尤其是英文版的图书，习惯于用大量篇幅对代码示例做各种讲解。我感觉这有些啰唆，想必很少有人去读第二遍，大家最多也就是用记号笔画出重点而已。既然如此，我们为何不信读者能阅读并理解这些代码呢？这本来就是程序员吃饭的本钱，最多在关键位置辅以注释便可。当然，阅读前提怕是要设定为非入门读者。好在我一再强调自己写的是第二本书，或曰“闲书”。

在本书中，对于理论层面，我会尝试说得明白些。当然，书中还会引入一些类比，这些类比或许不是非常合适，但却可以加深读者对相关问题的理解，毕竟不是所有人都能明白那些云里雾里的抽象理念。一如上面所言，文字与代码相辅相成，我们应静下心来用代码去验证文字背后的含义。在我眼里，代码也是一种自然语言，缩排跳转仿若图形，本就是最好的笔记注释。起码它离机器语言上有些距离，是为了便于人类阅读而发明的。

无论我说得多悦耳动听，这终归只是一本学习笔记，算不上专业，仅适合读者闲暇时翻阅一二。

关于本书

全套书分为上下两卷。上卷以语言为主，基本涵盖语言相关内容，包括语法、测试、调试，乃至解释器等层面的基本知识。下卷计划以标准库、优秀扩展库、并发编程，以及架构设计展开，算是对上卷“闲书”稍加修正。

书中示例运行环境：macOS 10.12，CPython 3.6，IPython 6.2

> 鉴于不同运行环境的差异性，示例输出结果（尤其是 id、内存地址等信息）会有所不同。另外，为阅读方便，本书对输出结果做了裁剪处理，请以实际运行结果为准。

读者定位

本书着重于剖析语言的相关背景和实现方式，适合有一定 Python 编程基础的读者（比如准备从 Python 2.7 升级到 Python 3.6 环境的读者）阅读。至于初学者，建议寻找从零开始、循序渐进地介绍如何编写代码的其他图书为佳。

联系方式

鄙人能力有限，书中难免存在错漏之处。读者如在阅读过程中发现任何问题，请与我联系，以便更正。谢谢！

- 邮件：qyuhen@hotmail.com
- 微博：weibo.com/qyuhen

雨 痕

二〇一七年，仲秋

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **提交勘误**：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/33274

上卷 语言详解

基于 Python 3.6

目录

第 1 章 概述 1

第 2 章 类型 5

2.1 基本环境 5

2.1.1 印象 5

2.1.2 名字 9

2.1.3 内存 16

2.1.4 编译 26

2.1.5 执行 30

2.2 内置类型 34

2.2.1 整数 35

2.2.2 浮点数 44

2.2.3 字符串 50

2.2.4 字节数组 61

2.2.5 列表 65

2.2.6 字典 76

2.2.7 集合 85

第 3 章 表达式 91
3.1 词法 91
3.1.1 源文件 92
3.1.2 代码 95
3.2 赋值 100
3.2.1 增量赋值 101
3.2.2 序列解包 102
3.2.3 作用域 107
3.3 运算符 110
3.3.1 链式比较 113
3.3.2 切片 114
3.3.3 逻辑运算 117
3.4 控制流 121
3.4.1 选择 121
3.4.2 循环 123
3.5 推导式 128
第 4 章 函数 133
4.1 定义 133
4.2 参数 141
4.3 返回值 149
4.4 作用域 151
4.5 闭包 157
4.6 调用 165
第 5 章 迭代器 174
5.1 迭代器概述 174

5.2 生成器 179
5.3 模式 186
5.4 函数式编程 190

第 6 章 模块 195

6.1 定义 195
6.2 导入 199
6.2.1 搜索 200
6.2.2 编译 202
6.2.3 引用 204
6.3 包 213
6.3.1 初始化 214
6.3.2 相对导入 218
6.3.3 拆分 221

第 7 章 类 223

7.1 定义 223
7.2 字段 230
7.3 属性 235
7.4 方法 238
7.5 继承 243
7.5.1 统一类型 244
7.5.2 初始化 246
7.5.3 覆盖 247
7.5.4 多继承 248
7.5.5 抽象类 254
7.6 开放类 256

7.7 运算符重载 263

第 8 章 异常 269

8.1 异常概述 269

8.2 断言 284

8.3 上下文 288

第 9 章 元编程 294

9.1 装饰器 294

9.1.1 实现 295

9.1.2 应用 301

9.2 描述符 304

9.3 元类 308

9.3.1 自定义 309

9.3.2 应用 314

9.4 注解 315

第 10 章 进阶 318

10.1 解释器 318

10.1.1 字节码 318

10.1.2 全局锁 321

10.1.3 执行过程 326

10.1.4 内存分配 334

10.1.5 垃圾回收 343

10.2 扩展 349

10.2.1 ctypes 349

10.2.2 Cython 356

第 11 章 测试 364
11.1 单元测试 364
11.1.1 unittest 365
11.1.2 unittest.mock 374
11.1.3 coverage 383
11.2 性能测试 383
11.2.1 timeit 383
11.2.2 profile 385
11.2.3 line profiler 388
11.2.4 memory profiler 390
11.2.5 pympler 391
第 12 章 工具 396
12.1 调试器 396
12.2 包管理 410

第 1 章 概述

Python 是一门相当有趣的编程语言。

其始于 1989 年末，约莫而立之年，比许多程序员的年龄还要大。在这段漫长的时光里，它见证了 C++的兴盛和群雄大战，看到了 Java 的异军突起和如日中天，更有同类 Ruby 凭借 RoR 领一时风骚，还有习惯丢三落四却每每笑到最后的 VC 被同族怼得灰头土脸。

在这期间还发生过什么？面向对象、设计模式、多层架构、面向服务等数不清的概念和名词。以程序界的划代标准，这已然是一个早该供在 WiKi 里的老古董，偶尔被某个年纪大的前辈拉出来讲古。然而，历经世事变幻，一朵朵“白莲花”最终都免不了闹个“腹黑”收场。就连当年那些无敌论的吹鼓手，如今都成了大肆指摘的异见人士。

且不管风云如何，Python 活得很好，依旧占据排行榜前列，可见其不全然是一部遗存的程序设计编年史。Python 不仅能当“胶水”写工具脚本，还借着大数据、深度学习、机器学习和人工智能的风潮，一跃成为当红之选。

当然，事物总有两面性。一方面，我们能从雨后春笋般出现的新技术支持名单里找到它的身影；而另一面，身边似乎并无多少人去使用或关注它。它被推荐给孩子作为编程入门语言，也被专业人士用于特定场合，可恰恰在靠编程吃饭的程序员主战场上位置尴尬。其虽然有浩如烟海的第三方支持，但说起来似乎只有系统维护和网站应用。所有这些，让我们对这门语言既熟悉又陌生。

喜欢它的各有因由，批评的则火力集中。在各大社区里，不乏有人对其性能、语法，乃至千年话题全局锁大加指责，以烘托某种语言才是更好的选择。这源自部分开发人员，尤其是新人追求大而全的心理，缺乏理性定位。历史上，还从没有一种语言能包办所有应用，更不曾讨好过所有人。

作为应用语言，脑门还刻着简单和优雅字号，自然要支持各时期的主流编程范式，竭力涵盖各类应用范畴，还需大费周章将复杂封装并隐藏起来，以期换取惯常喜新厌旧的程序员垂青。所以，从中你能看到命令式、函数式、面向对象、面向切面等程序设计方式。这造就了其广泛的支持，也带来易学难精的后果。历史包袱出现在所有步入中年的技术身上，其中有操作系统，有浏览器，自然也有编程语言。呼吁某某减肥和变革的声音不绝于耳，而后是新生代迈着轻盈的步伐后来居上。

可换个角度看，正因为与时俱进和兼容并蓄，方能存活至今。那些特立独行的，反倒未必笑到最后。君不见，方兴未艾的各路 NoSQL 不但因功能单一而横遭嫌弃，还遭看似老朽的“革命对象” RDBMS 反戈一击。要么纯粹享受寂寞，要么广博得汇溪成海。以“色”娱人，能讨巧于一时，终难长久。人们总有个错觉，似乎新技术是凭空出现，是年轻人的主场。但实际上，其依托早已存世，或埋于地下，或束之高阁，且待时机。

Python 的简单和周全，降低了非专业人士的使用门槛，毕竟他们的精力不会放在语言身上。此时，广泛支持就成为优点，即便不了解面向对象，也可用面向过程写点什么。更何况，那令人咋舌的生态系统里，总能找到你需要却又无法实现的东西。相当有趣的是，很有些专业扩展库，恰恰是用程序员看不上眼的代码完成的。兴许，作者只是个数学家，或图形学方面的天才。是以，任何一种设计都有其出现的原因和存在的理由。

在国内，将 Python 当作主力编程语言的人群很有限，其影响力和热度甚至不如某些后来者。这固然有其自身的种种原因，可社区疲软也显而易见。不管境况如何，难得有这样一门能长久陪伴，且行事周全的语言用于工作和学习，大家须珍惜。

Python 3

如果你对 Python 3 的了解尚停留在数年以前，那是时候更新一下认知了。下面这样一段文字，或许可代表生态圈的主流态度。

> What Python version should I use with Django ?
>
> Python 3 is recommended. Django 1.11 is the last version to support Python 2.7. Support for Python 2.7 and Django 1.11 ends in 2020.
>
> Since newer versions of Python are often faster, have more features, and are better supported, the latest version of Python 3 is recommended.
>
> You don’t lose anything in Django by using an older release, but you don’t take advantage of the improvements and optimizations in newer Python releases.
>
> —— Django FAQ

最初，迁入 Python 3 的阻碍可能是某个扩展库不支持，这也是很多人的主要理由。但到了今天，在 Python 3 Readiness 所统计的最流行的 360 个包里，有超过 95%支持 Python 3。起码对于新项目，这已不是问题。

另一个理由，应该是 Python 3 早期那让人失望的性能。可自 2008 年发布，至今 9 年，期间经多个版本的优化改进，其性能改善良多，早已不再是“弱鸡”的形象。

尽管在多年前，官方将 Python 2.7 EOL（end of life）推迟到 2020 年。可晃悠至今，所余时间已不足三年。即便因某些原因再度推迟，那也不是新项目继续使用 2.7 的理由，因为不会再有 Python 2.8。

还有，Python 3 的 asyncio 已成为主流异步框架。众多 Web Framework、Database Driver 等都已提供支持，并获得更好的执行性能。至于那些新增的、改进的，被摒弃且不合时宜的，等等，都值得我们去了解和尝试一下。

Python 3 Readiness：http://py3readiness.org/

Python Speed Center：https://speed.python.org/

Python 3.6 & Performance：http://igordavydenko.com/talks/by-pycon-2017/

Python 2.7 Countdown：https://pythonclock.org/

PEP 373：http://legacy.python.org/dev/peps/pep-0373/

第 2 章　类型

2.1　基本环境

作为一种完全面向对象，且兼顾函数式的编程语言，Python 的复杂程度要远高出许多人的设想，诸多概念被隐藏在看似简单的代码背后。为了更好且更深入地理解相关规则，在讲述语言特征以前，我们需对世界背景做初步的了解。

2.1.1　印象

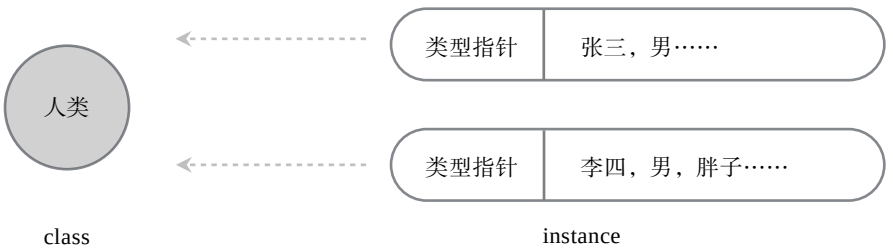
从抽象角度来看，每个运行中的程序（进程）都由数量众多的“鲜活”对象组成。每个对象都有其独特状态和逻辑，通过行为触发，或与其他对象交互来体现设计意图。面对众多个体，作为设计师自然要以宏观视角来规划世界。这里首先要做的，就是将所有个体分门别类归置于不同族群。

我们习惯将生物分为动物、植物，进而又有猫科、犬科等细分。通过对个体的研究，归纳其共同特征，抽象成便于描述的种族模板。有了模板后，可据此创建大量行为类似的个体。所以，分类是个基础工程。

在专业术语上，我们将族群或类别称作类型（class），将个体叫作实例（instance）。类型持有同族个体的共同行为和共享状态，而实例仅保存私有特性即可。如此，在内存空间

布局上才是最高效的。

> 以张三、李四为例。在创建抽象模型时，人类的共有特征，诸如吃饭、走路等，统统放到“人类”这个类型里。个体只保存姓名、性别、胖瘦、肤色这些即可。



每个实例都持有所属类型的指针。需要时，通过它间接访问目标即可。但从外在逻辑接口看，任何实例都是“完整”的。

存活实例对象都有“唯一”ID 值。

```
>>> id(123)
4487080432

>>> id("abc")
4488899864
```

> 不同 Python 实现使用不同算法，CPython 用内存地址作为 ID 值。这意味着它只能保证在某个时间，在存活对象里是唯一的。它不保证整个进程生命周期内的唯一，因为内存地址会被复用。因此，ID 就不适合作为全局身份标识。

可用 type 返回实例所属的类型。

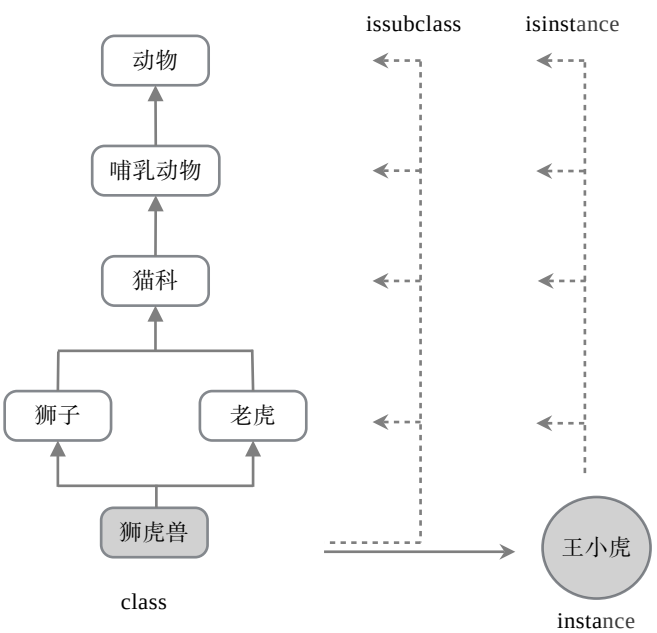
```
>>> type(1)
int

>>> type(1.2)
float
```

```
>>> type("hello")
str
```

要判断实例是否属于特定类型，可使用 isinstance 函数。

```
>>> isinstance(1, int)
True

>>> isinstance(1, float)
False
```

类型间可构成继承关系。就像老虎继承自猫科，而猫科又继承自哺乳动物，往上还有更顶层的类型。继承关系让类型拥有其所有祖先类型的特征。因历史原因，Python 允许多继承，也就是说可有多个父类型，好似人类同时拥有父族、母族遗传特征。



任何类型都是其祖先类型的子类，同样，对象也可以被判定为其祖先类型的实例。这与面向对象三大特性中的多态有关，后面再做详述。

```
class Animal: pass                    # 动物
```

```
class Mammal(Animal): pass            # 哺乳动物（继承自动物）
class Felidae(Mammal): pass           # 猫科
class Lion(Felidae): pass             # 狮子
class Tiger(Felidae): pass            # 老虎
class Liger(Lion, Tiger): pass        # 狮虎兽（继承自两个直系父类型）
```

```
>>> issubclass(Liger, Lion)
True

>>> issubclass(Liger, Tiger)
True

>>> issubclass(Liger, Animal)         # 是任何层次祖先类型的子类
True
```

```
>>> wxh = Liger()                     # 王小虎（狮虎兽实例）

>>> isinstance(wxh, Lion)
True

>>> isinstance(wxh, Tiger)
True

>>> isinstance(wxh, Animal)           # 是任何祖先类型的实例
True
```

事实上，所有类型都有一个共同祖先类型 object，它为所有类型提供原始模板，以及系统所需的基本操作方式。

```
>>> issubclass(Liger, object)
True

>>> issubclass(int, object)
True
```

类型虽然是抽象族群概念，但在实现上也只是个普通的对象实例。区别在于，所有类型都是由 type 创建的，这与继承无关。

```
>>> id(int)
4486772160
```

```
>>> type(int)
<class 'type'>

>>> isinstance(int, type)
True
```

怎么理解呢？单就类型对象而言，其本质就是用来存储方法和字段成员的特殊容器，用同一份设计来实现才是正常思路。

这就好比扑克牌，从玩法逻辑上看，J、Q、K、A 等都有不同含义。但从材质上看，它们完全相同，没道理用不同材料去制作这些内容不同的卡片。同理，继承也只是说明两个类型在逻辑上存在关联关系。如此，所有类型对象都属于 type 实例就很好理解了。

无论是编码，还是设计，都要正确区分逻辑与实现的差异。

```
>>> type(int)
<class 'type'>

>>> type(str)
<class 'type'>

>>> type(type)
<class 'type'>
```

当然，类型对象属于创建者这样的特殊存在。默认情况下，它们由解释器在首次载入时自动生成，生命周期与进程相同，且仅有一个实例。

```
>>> type(100) is type(1234) is int    # 指向同一类型对象
True
```

2.1.2 名字

在通常认知里，变量是一段具有特定格式的内存，变量名则是内存别名。因为在编码阶段，无法确定内存的具体位置，故使用名称符号代替。

> 注意变量名与指针的不同。

接下来，静态编译和动态解释型语言对于变量名的处理方式完全不同。静态编译器或链接器会以固定地址，或直接、间接寻址指令代替变量名。也就是说变量名不参与执行过程，可被剔除。但在解释型动态语言里，名字和对象通常是两个运行期实体。名字不但有自己的类型，还需分配内存，并介入执行过程。甚至可以说，名字才是动态模型的基础。

如果将内存寻址比喻成顾客按编号直接寻找服务柜台，那么名字就是一个接待员。任何时候，顾客都只能通过他间接与目标服务互动。从表面看，这似乎是高级会员待遇，但实际增加了中间环节和额外开销，于性能不利。但好处是，接待员与服务之间拥有更多的可调整空间，可用来增加代理和安全机制，甚至为缓存管理提供机会，具体可参考本书后续章节有关元编程的内容。

当然，名字必须与目标对象关联起来才有意义。

```
>>> x
NameError: name 'x' is not defined
```

最直接的关联操作就是赋值，而后对名字的引用都被解释为对目标对象进行操作。

```
>>> x = 100
>>> x
100

>>> x += 2
>>> x
102
```

赋值步骤：

1. 准备好右值目标对象（100）。

2. 准备好名字（x）。

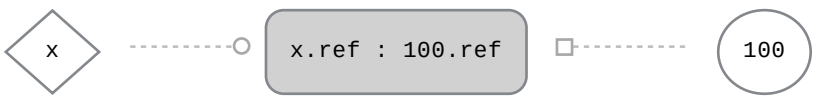
3. 在名字空间里为两者建立关联（namespace{x : 100}）。

即便如此，名字与目标对象之间也仅是引用关联。名字只负责找人，但对于此人一无所知。鉴于在运行期才能知道名字引用的目标类型，所以说 Python 是一种动态类型语言。

> Names have no type, but objects do.

名字空间

名字空间（namespace）是上下文环境里专门用来存储名字和目标引用关联的容器。



对 Python 而言，每个模块（源码文件）都有一个全局名字空间。而根据代码作用域，又有当前名字空间或本地名字空间一说。如果直接在模块级别执行，那么当前名字空间和全局名字空间相同。但在函数内，当前名字空间就专指函数作用域。

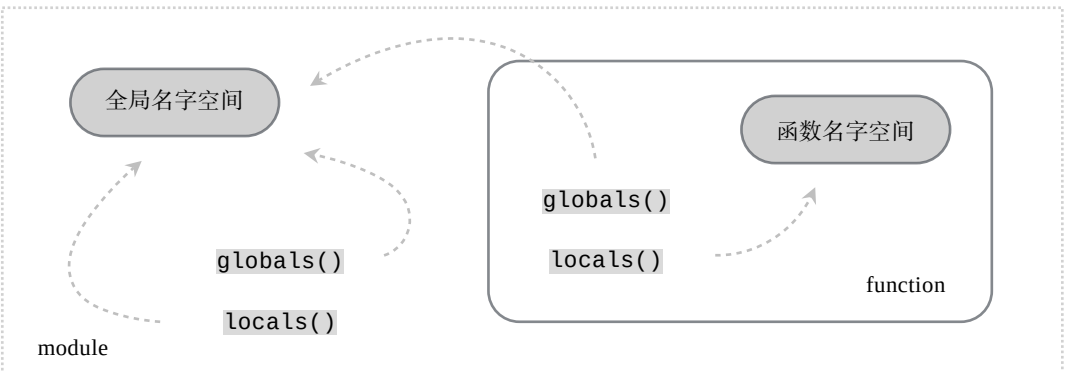
> 名字空间默认使用字典（dict）数据结构，由多个键值对（key/value）组成。
>
> 内置函数 globals 和 locals 分别返回全局名字空间和本地名字空间字典。

```
>>> x = 100

>>> id(globals())                          # 在模块作用域调用
4344493328

>>> id(locals())                           # 比对 ID 值，可以看到在模块级别，两者相同
4344493328

>>> globals()                              # 模块作用域的名字和对象映射关系
{'x': 100, ...}
```

```
>>> def test():
```

```
        x = "hello, test!"
        print(locals())                     # 指向函数本地名字空间
        print("locals:", id(locals()))
        print("globals:", id(globals()))    # 指向全局名字空间

>>> test()                                  # 在函数作用域调用
{'x': 'hello, test!'}                       # 此时，locals 输出函数名字空间
locals : 4347899696                         # locals != globals
globals: 4344493328
```

可见，globals 总是固定指向模块名字空间，而 locals 则指向当前作用域环境。



在初步了解后，我们甚至可直接修改名字空间来建立关联引用。这与传统变量定义方式有所不同。

```
>>> globals()["hello"] = "hello, world!"
>>> globals()["number"] = 12345

>>> hello
'hello, world!'

>>> number
12345
```

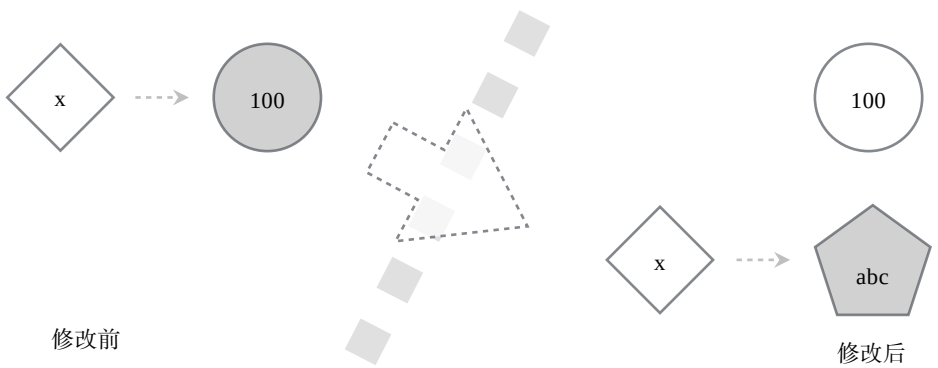
并非所有时候都能直接操作名字空间。函数执行使用缓存机制，直接修改本地名字空间未必有效。在正常编码时，应尽可能避免直接修改名字空间。

在名字空间字典里，名字只是简单字符串主键，其自身数据结构里没有任何目标对象信息。通过名字访问目标对象，无非是以名字为主键去字典里读取目标对象指针引用。也正因如此，名字可重新关联另一对象，完全不在乎其类型是否与前任相同。

```
>>> x = 100

>>> x
100

>>> id(x)
4319528720

>>> globals()
{'x': 100, ...}
```

```
>>> x = "abc"                    # 重新关联其他对象，没有类型限制

>>> x
'abc'

>>> id(x)                        # 通过输出 id，可以看到关联新对象，而非修改原对象内容
4321344792

>>> globals()                    # 名字空间里的引用变更
{'x': 'abc', ...}
```

赋值操作仅是让名字在名字空间里重新关联，而非修改原对象。



和一个名字只能引用一个对象不同，单个对象可以同时有多个名字，无论是在相同或不同的名字空间里。

> 一个人在办公室里（名字空间）的名字可以是老王、王大虎，等等，这时，这个人在单一空间里有多个名字。出了办公室，在全公司（另一名字空间）范围内，他还可有王处长等其他名字。

```
>>> x = 1234
>>> y = x                          # 总是按引用传递

>>> x is y                         # 判断是否引用同一对象
True

>>> id(x)                          # 输出 id 值来确认上述结论
4350620272

>>> id(y)
4350620272

>>> globals()                      # 1234 有两个名字
{'x': 1234, 'y': 1234, ...}
```

必须使用 is 判断两个名字是否引用同一对象。相等操作符并不能确定两个名字指向同一对象，这涉及操作符重载，或许仅用来比较值是否相等。

```
>>> a = 1234                       # 请使用大数字。因为小数字常量会被缓存复用
>>> b = 1234

>>> a is b                         # 指向不同对象
False

>>> a == b                         # 但值相等
True
```

命名规则

名字应选用有实际含义，且易于阅读和理解的字母或单词组合。

- 以字母或下画线开头。
- 区分大小写。
- 不能使用保留关键字（reserved word）。

```
>>> x = 1
>>> X = "abc"

>>> x is X                # 区分大小写
False
```

为统一命名风格，建议：

- 类型名称使用 CapWords 格式。
- 模块文件名、函数、方法成员等使用 lower_case_with_underscores 格式。
- 全局常量使用 UPPER_CASE_WITH_UNDERSCORES 格式。
- 避免与内置函数或标准库的常用类型同名，因为这样易导致误解。

```
>>> OK = 200
>>> KEY_MIN = 10

>>> set_name = None
```

尽管 Python 3 支持用中文字符作为名字，但这并不是好选择。

保留关键字如下：

```
False      class      finally    is         return
True       continue   for        lambda     try
None       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
assert     else       import     pass
break      except     in         raise
```

提示：Python 2.x 里的 exec、print，在 Python 3.x 里已变成函数，不再是保留字。

```
>>> class = 100
SyntaxError: invalid syntax
```

如要检查动态生成代码是否违反保留字规则，可用 keyword 模块。

```
>>> import keyword

>>> keyword.kwlist
['False', 'None', ... 'with', 'yield']
```

```
>>> keyword.iskeyword("is")
True

>>> keyword.iskeyword("print")
False
```

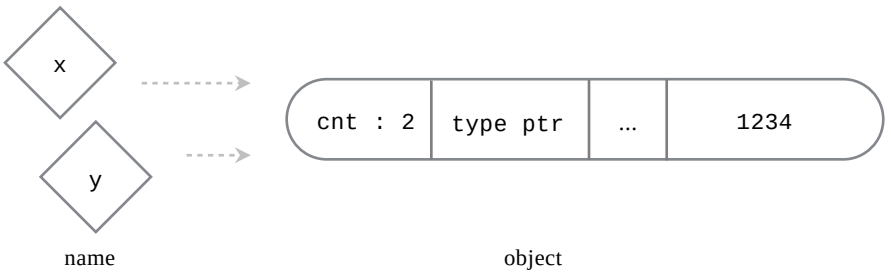
以下画线开头的名字，代表特殊含义。

- 模块成员以单下画线开头（_x），属私有成员，不会被星号导入。
- 类型成员以双下画线开头，但无结尾（__x），属自动重命名私有成员。
- 以双下画线开头和结尾（__x__），通常是系统成员，应避免使用。
- 在交互模式下，单下画线（_）返回最后一个表达式结果。

```
>>> 1 + 2 + 3
6

>>> _
6
```

2.1.3 内存

Python 里没有值类型、引用类型之分。事实上，每个对象都很重。即便是简单的数字，也有标准对象头，以及保存类型指针和引用计数等信息。如果是字符串、列表这类变长类型，还要记录数据项的长度，然后才是状态数据。




```
>>> x = 1234

>>> import sys

>>> sys.getsizeof(x)            # Python 3 里的 int 也是变长结构
28
```

总是通过名字来完成“引用传递”（pass-by-reference）。名字关联会增加计数，反之会减少计数。如删除全部名字关联，那么该对象的引用计数归零，会被系统回收。这就是默认的引用计数垃圾回收机制（Reference Counting Garbage Collector）。

关于 Python 是 pass-by-reference，还是 pass-by-value，有不同说法。究其原因，还是名字不是内存符号造成的。如果变量不包括名字所关联的目标对象，那么就是值传递。因为传递是通过复制名字关联来实现的，类似于复制指针，或许正确的说法是 pass-by-object-reference。不过在编码时，我们关注的是目标对象，而非名字自身。从这点上说，引用传递更能清晰地解释代码的实际意图。

基于立场不同，对某些问题会有不同的理论解释。有时候，反过来用实现来推导理论，或许能加深理解，更好地掌握相关知识。

```
>>> a = 1234
>>> b = a

>>> sys.getrefcount(a)          # getrefcount 也会通过参数引用目标对象，导致计数+1
3

>>> del a                       # 删除其中一个名字，减少计数

>>> sys.getrefcount(b)
2
```

所有对象都由内存管理系统在特定区域统一分配。无论是赋值、传参，还是返回局部变量都无须关心内存位置，并没有逃逸或隐式复制行为发生。

基于性能考虑，像 Java、Go 这类语言，编译器优先在栈上分配对象内存。但考虑到闭包、接口、外部引用等因素，原本在栈上分配的对象可能会“逃逸”到堆上。这势必会延长对象生命周期，加大垃圾回收负担。所以，会有专门的逃逸分析（escape analysis），用于代码和算法优化。

Python 解释器虽然也有执行栈，但不会在栈上为对象分配内存。可以认为所有原生对象（非 C、Cython 等扩展）都在“堆”上分配。

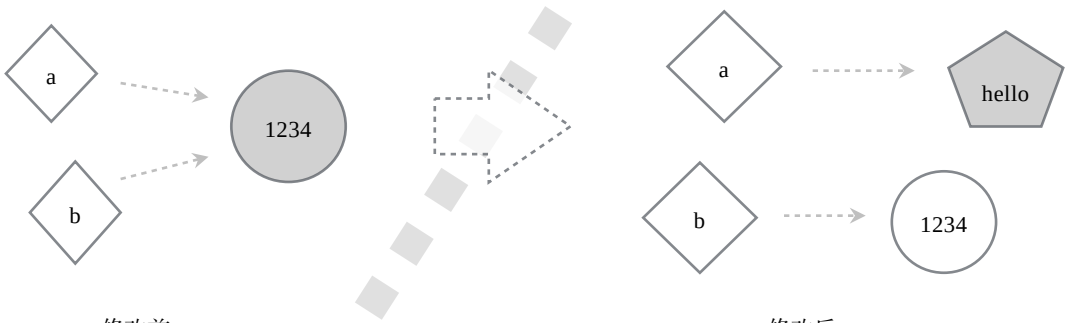
```
>>> a = 1234

>>> def test(b):
        print(a is b)              # 参数 b 和 a 都指向同一对象

>>> test(a)
True
```

```
>>> def test():
        x = "hello, test"
        print(id(x))
        return x                   # 返回局部变量

>>> a = test()                     # 对比 id 结果，确认局部变量被导出
4371868400

>>> id(a)
4371868400
```

将对象多个名字中的某个重新赋值，不会影响其他名字。

```
>>> a = 1234
>>> b = a

>>> a is b
True
>>> a = "hello"                    # 将 a 重新关联到字符串对象
```

```
>>> a is b                          # 此时，a、b 分别指向不同的对象
False

>>> a
'hello'

>>> b
1234
```



修改前　　修改后

```
>>> a = 1234

>>> def test(b):
        print(b is a)               # True；引用传递，此时 b 和 a 指向同一对象
        b = "hello"                 # 在 locals 中，将 b 重新关联到字符串
        print(b is a)               # False；此时 b 和 a 分道扬镳

>>> test(a)
True
False

>>> a                               # 显然对 a 没有影响
1234
```

注意，只有对名字赋值才会变更引用关系。如下面示例，函数仅仅是通过名字引用修改了列表对象自身，而并未重新关联到其他对象。

```
>>> a = [1, 2]

>>> def test(b):
        b.append(3)                 # 通过名字 b 向列表追加数据
```

```
        print(b)                    # 查看修改结果

>>> test(a)
[1, 2, 3]

>>> a                               # a 和 b 指向同一对象，自然也能输出“同步”结果
[1, 2, 3]
```

弱引用

如果说，名字与目标对象关联构成强引用关系，会增加引用计数，进而影响其生命周期，那么，弱引用（weak reference）就是减配版，其在保留引用前提下，不增加计数，也不阻止目标被回收。

> 不是所有类型都支持弱引用，比如 int、tuple 等。

```
class X:
    def __del__(self):              # 析构方法，用来观察实例回收
        print(id(self), "dead.")
```

```
>>> a = X()                         # 创建实例

>>> sys.getrefcount(a)
2
```

```
>>> import weakref

>>> w = weakref.ref(a)              # 创建弱引用

>>> w() is a                        # 通过弱引用访问目标对象
True

>>> sys.getrefcount(a)              # 弱引用并未增加目标对象引用计数
2
```

```
>>> del a                           # 解除目标对象名字引用，对象被回收
4384434048 dead.
```

```
>>> w() is None                    # 弱引用失效
True
```

标准库另有一些弱引用相关函数，以及弱引用字典、集合等容器。

```
>>> a = X()
>>> w = weakref.ref(a)

>>> weakref.getweakrefcount(a)
1

>>> weakref.getweakrefs(a)
[<weakref at 0x10548f778; to 'X' at 0x10553b668>]

>>> hex(id(w))
'0x10548f778'
```

弱引用可用于一些特定场合，比如缓存、监控等。这类“外挂”场景不应该影响目标对象，不能阻止它们被回收。弱引用的另一个典型应用是实现 Finalizer，也就是在对象被回收时执行额外的“清理”操作。

```
>>> a = X()

>>> def callback(w):
        print(w, w() is None)

>>> w = weakref.ref(a, callback)          # 创建弱引用时设置回调函数

>>> del a                                 # 回收目标对象时，回调函数被调用
4384343488 dead.
<weakref at 0x1057f2818; dead> True
```

为什么不使用析构方法（__del__）？

很简单，析构方法作为目标成员，其用途是完成对象内部资源清理。它无法感知，也不应该处理与之无关的外部场景。但在实际开发中，外部关联场景有很多，那么用 Finalizer 才是合理设计，因为这样只有一个不会侵入的观察员存在。

> 注意，回调函数参数为弱引用而非目标对象。回调函数执行时，目标已无法访问。

抛开对生命周期的影响不说，弱引用与普通名字的最大区别在于其类函数的调用语法。不过可用 proxy 改进，使其和名字引用语法保持一致。

```
>>> a = X()
>>> a.name = "Q.yuhen"

>>> w = weakref.ref(a)

>>> w.name
AttributeError: 'weakref' object has no attribute 'name'

>>> w().name                            # 必须使用调用语法
'Q.yuhen'
```

```
>>> p = weakref.proxy(a)
>>> p
<__main__.X at 0x1055ebe58>

>>> p.name                              # 直接访问目标成员
'Q.yuhen'

>>> p.age = 60                          # 通过 proxy 直接向目标对象添加或设置成员
>>> a.age                               # 验证目标对象属性
60

>>> del a                               # 同样不会影响目标生命周期
4387316960 dead.
```

对象复制

从初学者角度看，基于名字的引用传递要比值传递自然得多。试想，在日常生活中，谁会因为名字被别人呼来唤去就莫名其妙克隆出一个自己来？但在有经验的程序员眼里，事情恐怕得反过来。

在调用函数时，我们或许仅仅是想传递一个状态，而非整个实体。这好比把自家姑娘的

生辰八字，外加一幅美人图交给媒人，而断断没有让媒人直接把人领走的道理。另一方面，并发编程的多个执行单元共享实例，会引起数据竞争（data race），也是一个大麻烦。

对象控制权该由创建者负责，而不能寄希望于接收参数的被调用方。必要时，可以用不可变类型，或对象的复制品作为参数传递。

> 不可变类型包括 int、float 、str、bytes、tuple、frozenset 等。因这些类型不能改变状态，所以被多方只读引用不会有什么问题。至于对象复制，除自带复制方法的类型外，还可使用标准库的 copy 模块，或 pickle、json 等序列化（object serialization）方案。

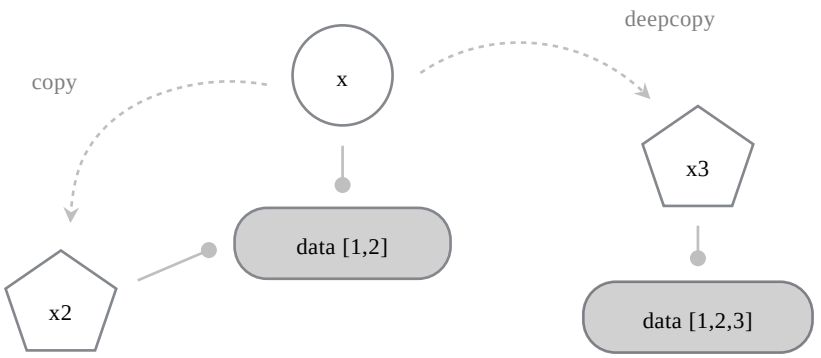
复制分浅拷贝（shallow copy）和深度拷贝（deep copy）两种。对于对象内部成员，浅拷贝仅复制名字引用，而深度拷贝会递归复制所有引用成员。

```
>>> class X: pass
>>> x = X()                          # 创建实例
>>> x.data = [1, 2]                  # 成员 data 引用一个列表
```

```
>>> import copy

>>> x2 = copy.copy(x)                # 浅拷贝

>>> x2 is x                          # 复制成功
False

>>> x2.data is x.data                # 但成员 data 依旧指向原列表，仅仅复制了引用
True
```

```
>>> x3 = copy.deepcopy(x)            # 深度拷贝

>>> x3 is x                          # 复制成功
False

>>> x3.data is x.data                # 成员 data 列表同样被复制
False

>>> x3.data.append(3)                # 向复制的 data 列表追加数据

>>> x3.data
```

```
[1, 2, 3]

>>> x.data                          # 没有影响原列表
[1, 2]
```



相比于复制，序列化将对象转换成可存储和传输的格式，反序列化则逆向还原。至于格式，可以是 pickle 二进制，也可以是 JSON、XML 等文本格式。二进制拥有最好的性能，但从数据传输和兼容性来看，JSON 可能更佳。

二进制序列化还可选择 MessagePack 等跨平台第三方解决方案。

```
>>> class X: pass

>>> x = X()
>>> x.str = "string"
>>> x.list = [1, 2, 3]
```

```
>>> import pickle

>>> d = pickle.dumps(x)             # 序列化，返回二进制字节序列
>>> x2 = pickle.loads(d)            # 反序列化，返回新的 X 实例

>>> x2 is x
False

>>> x2.list is x.list               # 基于字节序列构建，与原对象再无关系
False
```

无论哪种方案都存在一定限制，具体以相关文档为准。

循环引用垃圾回收

引用计数机制实现简单，能实时响应，在计数归零时立即清理该对象所占的内存，绝大多数时候都能高效运作。只是当两个或更多对象构成循环引用（reference cycle）时，该机制就会遭遇麻烦。因为彼此引用导致计数永不归零，从而无法触发回收操作，形成内存泄漏。为此，另有一套专门用来处理循环引用的垃圾回收器（以下简称 gc）作为补充。

> 单个对象也能构成循环引用，比如列表把自身引用作为元素存储。

相比于在后台默默工作的引用计数，这个可选垃圾回收器拥有更多控制选项，包括将其临时或永久关闭的选项。

```
class X:
    def __del__(self):
        print(self, "dead.")
```

```
>>> import gc
>>> gc.disable()                              # 关闭 gc

>>> a = X()
>>> b = X()

>>> a.x = b                                   # 构建循环引用
>>> b.x = a

>>> del a                                     # 删除全部名字后，对象并未被回收，引用计数失效
>>> del b
```

```
>>> gc.enable()                               # 重新启用 gc

>>> gc.collect()                              # 主动启动一次回收操作，循环引用对象被正确回收
<__main__.X object at 0x1009d9160> dead.
<__main__.X object at 0x1009d9128> dead.
```

> 虽然可用弱引用打破循环，但实际开发时很难这么做。就本例而言，a.x 和 b.x 都需要保证目标存活，这是逻辑需求，但弱引用无法保证这点。另外，循环引用可能由很多对象因复杂流程间接造成，极难被发现，自然也就无法提前使用弱引用方案。
>
> 在早期版本里，gc 不能回收包含__del__的循环引用，但现在这已不是问题。另外，IPython 对于弱引用和垃圾回收存在干扰，建议用原生交互环境或源码文件测试本节代码。

在解释器启动时，gc 默认被打开，并跟踪所有可能造成循环引用的对象。相比于引用计数，它是一种延迟回收方式，只有当内部预设的阈值条件满足时，才会在后台启动。虽然可忽略该条件，强制执行回收，但不建议频繁使用。具体细节，可参考本书后续内容。

对于某些性能优先的算法，在确保没有循环引用的前提下，临时关闭 gc 可获得更好的性能。甚至于某些极端优化策略里，会完全屏蔽垃圾回收，以重启进程来回收资源。

> 做性能测试（timeit）时，会关闭 gc，避免垃圾回收对执行计时造成影响。

2.1.4 编译

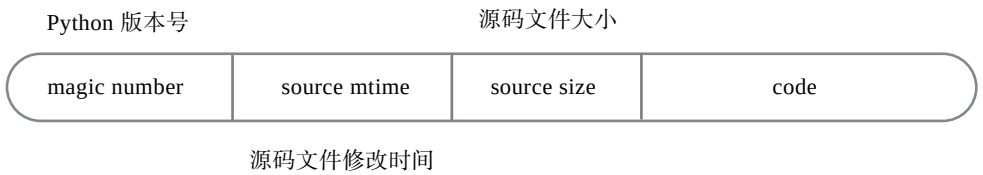
源码须先编译成字节码，才能交由解释器以解释方式执行。这也是 Python 性能为人诟病的一个重要原因。

> 字节码（byte code）是中间代码，面向后端编译器或解释器。要么解释执行，要么二次编译成机器代码（native code）执行。字节码指令通常基于栈式虚拟机（stack-based VM）实现，没有寄存器等复杂概念，实现简单。且其具备中立性，与硬件架构、操作系统等无关，便于将编译与平台实现分离，是跨平台语言的主流方案。
>
> 除官方 CPython 外，也可尝试内置即时编译（Just-In-Time compiler）的版本（PyPy）。它们可提供更好的执行性能，以及更少的内存占用。只是标准有些滞后，或不支持全部语言特性。

除了交互模式和手工编译，正常情况下，源码文件在被导入（import）时完成编译。编

译后的字节码数据被缓存复用，通常还会保存到硬盘。

Python 3 使用专门目录保存字节码缓存文件（__pycache__/*.pyc ）。这样在程序下次启动时，可避免再次编译，以提升导入速度。缓存文件头部存储了编译的相关信息，用来判断源码文件是否更新，是否需要重新编译。



> 下载 Python 3.6 源码，阅读 Lib/importlib/text_bootstrap_external.py 文件里的_code_to_bytecode 代码，可看到字节码头部信息的构成。该文件还有“Finder/loader utility code”注释，对相关格式和内容做了更详细的解释。

除作为执行指令的字节码外，还有很多元数据，共同组成执行单元。从这些元数据里，可以获得参数、闭包等诸多信息。

```
def add(x, y):
    return x + y
```

```
>>> add.__code__
<code object add at 0x1070ce9c0>

>>> dir(add.__code__)
['co_argcount', 'co_code', 'co_consts', ... 'co_varnames']

>>> add.__code__.co_varnames
('x', 'y')

>>> add.__code__.co_code
b'|\x00|\x01\x17\x00S\x00'
```

就像我们无法阅读机器码一样，对于字节码指令，同样需要借助一点手段。所幸，标准库提供了一个 dis 模块，让我们可以直接“反汇编”（ disassembly ）。

```
>>> import dis

>>> dis.dis(add)
  2           0 LOAD_FAST                0 (x)
              2 LOAD_FAST                1 (y)
              4 BINARY_ADD
              6 RETURN_VALUE
```

相比于 x86-64、ARM 等汇编指令，字节码反汇编结果非常易读。标准库帮助文档更是对所有指令做了详细说明。与隐藏大量细节的 Python 语言不同，字节码指令能直接体现解释器的执行流程和细节。在后续章节，我们会大量使用该方式来验证语言特征。

> 本书的第 10 章会对解释器、反汇编等做出详细解释，现在只需有初步了解即可。

某些时候，需要手工完成编译操作。

```
source = """
    print("hello, world!")
    print(1 + 2)
"""
```

```
>>> code = compile(source, "demo", "exec")        # 编译，提供一个文件名便于输出提示

>>> dis.show_code(code)                           # 输出相关信息
Filename:          demo
Constants:
   0: 'hello, world!'
   1: 1
   2: 2
   3: None
   4: 3
Names:
   0: print

>>> dis.dis(code)
  2           0 LOAD_NAME                0 (print)
              2 LOAD_CONST               0 ('hello, world!')
              4 CALL_FUNCTION            1
              6 POP_TOP
```

```
  3           8 LOAD_NAME                0 (print)
             10 LOAD_CONST               4 (3)
             12 CALL_FUNCTION            1
             14 POP_TOP
             16 LOAD_CONST               3 (None)
             18 RETURN_VALUE
```

```
>>> exec(code)
hello, world!
3
```

除内置 compile 函数外，标准库还有编译源码文件的相关模块。可在代码中调用，或直接在命令行执行。

> 可只发布 pyc 文件，算是对源码的简单保护。详情参考第 6 章的相关内容。

```
>>> import py_compile, compileall

>>> py_compile.compile("main.py")
'__pycache__/main.cpython-36.pyc'

>>> compileall.compile_dir(".")
Listing '.'...
Compiling './test.py'...
True
```

```
$ python -m compileall .
```

2.1.5 执行

除预备好的代码外，还可在运行期动态执行“未知”代码。当然，这并不局限于从网络下载新的功能模块，其还常被用来实现动态生成设计。

以标准库 namedtuple 为例，它基于字符串模板在运行期构建新的类型。类似的还有 ORM 框架，通过读取数据表结构动态生成数据模型和映射关系，移除了烦琐且看上去不太美观的架构（schema）定义代码。

不同于专业人员编写的插件，动态执行逻辑可能来源于用户输入（比如计算公式），也可能是由运维人员后台推送（比如新的加密方式或数据格式）的。从这点上来说，它为架构设计和功能扩展带来了更大的灵活性。

> 动态执行和语言并无直接关系。就算是 Java、C#这类静态语言，也可通过 CodeDOM、Emit 等方式动态生成、编译和执行代码。

```
>>> import collections

>>> User = collections.namedtuple("User", "name,age", verbose=True)   # 动态生成 User 类型

class User(tuple):
    _fields = ('name', 'age')
    ...
```

```
>>> User
__main__.User

>>> u = User("Q.yuhen", 60)

>>> u
User(name='Q.yuhen', age=60)
```

且不管代码如何生成，最终要么以模块导入执行，要么调用 eval、exec 函数执行。这两个内置函数使用简单，eval 执行单个表达式，exec 应对代码块，接受字符串或已编译好的代码对象（code）作为参数。如果是字符串，就会检查是否符合语法规则。

```
>>> s = input()                          # 输入表达式，比如数学运算
(1 + 2) * 3

>>> eval(s)                              # 执行表达式
9
```

```
>>> s = """
    def test():
        print("hello, world!")

    test()
```

```
"""

>>> exec(s)                                # 执行多条语句
hello, world!
```

无论选择哪种方式执行，都必须有相应的上下文环境。默认直接使用当前全局和本地名字空间。如同普通代码那样，从中读取目标对象，或写入新值。

```
>>> x = 100

>>> def test():
        y = 200
        print(eval("x + y"))            # 从上下文名字空间读取 x、y 值

>>> test()
300
```

```
>>> def test():
        print("test:", id(globals()), id(locals()))
        exec("print('exec:', id(globals()), id(locals()))")     # 对比名字空间的 id 值

>>> test()
test: 4471161768 4468227456
exec: 4471161768 4468227456
```

有了操作上下文名字空间的能力，动态代码就可向外部环境注入新的成员，比如说构建新的类型，或导入新的算法逻辑等，最终达到将动态逻辑或其结果融入，成为当前体系组成部分的设计目标。

```
>>> s = """
    class X: pass

    def hello():
        print("hello, world!")
"""

>>> exec(s)                          # 执行后，会在名字空间内生成 X 类型和 hello 函数
```

```
>>> X
__main__.X
```

```
>>> X()
<__main__.X at 0x10aafe8d0>

>>> hello()
hello, world!
```

某些时候，动态代码来源不确定，基于安全考虑，必须对执行过程进行隔离，阻止其直接读写环境数据。如此，就须显式传入容器对象作为动态代码的专用名字空间，以类似简易沙箱（sandbox）方式执行。

根据需要，分别提供 globals、locals 参数，也可共用同一名字空间字典。

为保证代码正确执行，解释器会自动导入__builtins__模块，以便调用内置函数。

```
>>> g = {"x": 100}
>>> l = {"y": 200}

>>> eval("x + y", g, l)                    # 为动态代码 globals、locals 分别指定字典
300
```

```
>>> ns = {}

>>> exec("class X: pass", ns)              # globals、locals 共用同一字典

>>> ns
{'X': X,  '__builtins__': ...}
```

同时提供两个名字空间参数时，默认总是 locals 优先，除非在动态代码里明确指定使用 globals。这涉及多层次名字搜索规则 LEGB。在本书的后续内容里，你会看到搜索规则和名字空间一样无处不在，有巨大的影响力。

```
>>> s = """
    print(x)                # locals.x

    global y                # globals.y
    y += 100
```

```
    z = x + y                # locals.z = l.x + g.y
"""
```

```
>>> g = {"x": 10, "y": 20}
>>> l = {"x": 1000}

>>> exec(s, g, l)
1000

>>> g
{'x': 10, 'y': 120}

>>> l
{'x': 1000, 'z': 1120}
```

前面曾提及，在函数作用域内，locals 函数总是返回执行栈帧（stack frame）名字空间。因此，即便显式提供 locals 名字空间，也无法将其注入动态代码的函数内。

```
>>> s = """
    print(id(locals()))              # 我们提供的 locals 参数

    def test():
        print(id(locals()))          # 函数调用栈帧名字空间

    test()
"""
```

```
>>> ns = {}
>>> id(ns)
4473689720

>>> exec(s, ns, ns)                 # 显然，test.locals 和我们提供的 ns locals 不同
4473689720
4474406808
```

如果是动态下载的源码文件，可用标准库 runpy 导入执行。

2.2 内置类型

与自定义类型（user-defined）相比，内置类型（built-in）算是特权阶层。除了它们是复合数据结构的基本构成单元以外，最重要的是被编译器和解释器特别对待。比如，核心级别的指令和性能优化，专门设计的高效缓存，等等。

作为“自带电池”（batteries included）的 Python，在这点上可谓丰富之极。很多时候，只用基本数据类型就可完成相对复杂的算法逻辑，更不要说标准库里的那些后备之选了。

内置的基本数据类型，可简单分为数字、序列、映射和集合几类。另根据其实例是否可被修改，又有可变（mutable）和不可变（immutable）之别。

名　　称	分　　类	可 变 类 型
int	number	N
float	number	N
str	sequence	N
bytes	sequence	N
bytearray	sequence	Y
list	sequence	Y
tuple	sequence	N
dict	mapping	Y
set	set	Y
frozenset	set	N

标准库 collections.abc 列出了相关类型的抽象基类，可据此判断其基本行为方式。

```
>>> import collections.abc

>>> issubclass(str, collections.abc.Sequence)
True

>>> issubclass(str, collections.abc.MutableSequence)
False
```

在日常开发时，应优先选择内置类型（标准库）。除基本性能考虑外，还可得到跨平台兼容性，以及升级保障。轻易引入不成熟的第三方代码，会提升整体复杂度，增加潜在错误风险，更不利于后续升级和代码维护。

2.2.1 整数

Python 3 将原 int、long 两种整数类型合并为 int，采用变长结构。虽然这会导致更多的内存开销，但胜在简化了语言规则。

同样不再支持表示 long 类型的 L 常量后缀。

变长结构允许我们创建超大的天文数字，理论上仅受可分配内存大小的限制。

```
>>> x = 1

>>> type(x)
int
>>> sys.getsizeof(x)
28
```

```
>>> y = 1 << 10000

>>> y
199506311688075838488374216268358508382349683188619 24...04792596709376

>>> type(y)
int

>>> sys.getsizeof(y)
1360
```

从输出结果看，尽管都是 int 类型，但 x 和 y 所占用的内存大小差别巨大。在底层实现上，通过不定长结构体（variable length structure）按需分配内存。

对于较长数字，为便于阅读，习惯以千分位进行分隔标记。但逗号在 Python 语法中有特殊含义，故改用下画线表示，且不限分隔位数。

```
>>> 78,654,321              # 表示 tuple 语法，不能用作千分位表达
(78, 654, 321)
```

```
>>> 78_654_321
78654321

>>> 786543_21
78654321
```

除十进制外，数字还可以二进制（bin）、八进制（oct），以及十六进制（hex）表示。下画线分隔符号同样适用于这些进制的字面量。

> 二进制可用来设计类似位图（bitmap）这类开关标记类型，系统管理命令 chmod 使用八进制设置访问权限，至于十六进制则常见于反汇编等逆向操作。
>
> 八进制不再支持 012 这样的格式，只能以 0o（或大写）前缀开头。

```
>>> 0b110011          # bin
51

>>> 0o12              # oct
10

>>> 0x64              # hex
100

>>> 0b_11001_1
51
```

转换

可用内置函数将整数转换为指定进制字符串，或反向用 int 还原。

```
>>> bin(100)
'0b1100100'
```

```
>>> oct(100)
'0o144'

>>> hex(100)
'0x64'
```

int 函数默认为十进制，会忽略空格、制表符等空白符。如指定进制，可省略相关进制前缀。

```
>>> int("0b1100100", 2)
100

>>> int("0o144", 8)
100

>>> int("0x64", 16)
100

>>> int("64", 16)               # 省略进制前缀
100
```

```
>>> int(" 100 ")                # 忽略多余空白符
100

>>> int("\t100\t")
100
```

当然，用 eval 也能完成转换，无非是把字符串当作普通表达式而已。但相比于直接以 C 实现的转换函数，其性能要差很多，毕竟动态运行需要额外编译和执行开销。

```
>>> eval("0o144")
100
```

还有一种转换操作是将整数转换为字节数组，这常见于二进制网络协议和文件读写。在这里需要指定字节序（byte order），也就是常说的大小端（big-endian, little-endian）。

> 无论什么类型的数据，在系统底层均以字节方式存储。以整数 0x1234 为例，可分为两个字节，高位字节 0x12，低位 0x34。不同硬件架构会采取不同的存储顺序，高位在前（big-endian，大端）或低位在前（little-endian，小端），这与其设计有关。
>
> 日常使用较多的 Intel x86、AMD64 采用小端。ARM 则两种都支持，默认采用小端较多，可自行设定。另外，TCP/IP 网络字节序采用大端，这属于协议定义，与硬件架构和操作系统无关。

转换操作须指定目标字节数组大小，考虑到整数类型是变长结构，故通过二进制位长度计算。另外，通过 sys.byteorder 获取当前系统字节序。

```
>>> x = 0x1234
>>> n = (x.bit_length() + 8 - 1) // 8            # 计算按 8 位对齐所需的字节数

>>> b = x.to_bytes(n, sys.byteorder)

>>> b.hex()
'3412'

>>> hex(int.from_bytes(b, sys.byteorder))
'0x1234'
```

运算符

支持常见的数学运算，但要注意除法在 Python 2 和 Python 3 里的差异。

Python 2.7

```
>>> 3 / 2
1

>>> type(3 / 2)
<type 'int'>
```

Python 3.6

```
>>> 3 / 2
1.5
```

```
>>> type(3 / 2)
<class 'float'>
```

> 除法运算符分单斜线和双斜线两种。单斜线被称作 True Division，无论是否整除，总是返回浮点数。而双斜线 Floor Division 会截掉小数部分，仅返回整数结果。

```
>>> 4 / 2              # true division
2.0

>>> 3 // 2             # floor division
1
```

如要获取余数，可用取模运算符（mod）或 divmod 函数。

```
>>> 5 % 2
1

>>> divmod(5, 2)
(2, 1)
```

另一不同之处是，Python 3 不再支持数字和非数字类型的比较操作。

Python 2.7

```
>>> 1 > ""
False

>>> 1 < []
True
```

Python 3.6

```
>>> 1 > ""
TypeError: '>' not supported between instances of 'int' and 'str'

>>> 1 < []
TypeError: '<' not supported between instances of 'int' and 'list'
```

布尔

布尔是整数子类型，也就是说 True、False 可被当作数字直接使用。

```
>>> issubclass(bool, int)
True

>>> isinstance(True, int)
True
```

```
>>> True == 1
True

>>> True + 1
2
```

在进行布尔转换时，数字零、空值（None）、空序列和空字典都被视作 False，反之为 True。

```
>>> data = (0, 0.0, None, "", list(), tuple(), dict(), set(), frozenset())

>>> any(map(bool, data))
False
```

> 对于自定义类型，可通过重写__bool__或__len__方法影响 bool 转换结果。

枚举

Python 语言规范里没有枚举（enum）的定义，而是采用标准库实现。

在多数语言里，枚举是面向编译器，类似于数字常量的存在。但到了 Python 中，事情变得有些复杂。首先定义枚举类型，随后由内部代码生成枚举值实例。

```
>>> import enum

>>> Color = enum.Enum("Color", "BLACK YELLOW BLUE RED")
```

```
>>> isinstance(Color.BLACK, Color)
True
```

```
>>> list(Color)
[<Color.BLACK: 1>, <Color.YELLOW: 2>, <Color.BLUE: 3>, <Color.RED: 4>]
```

没有规定枚举值必须是整数。通过继承，可将其指定为任何类型。

```
>>> class X(enum.Enum):
        A = "a"
        B = 100
        C = [1, 2, 3]

>>> X.C
<X.C: [1, 2, 3]>
```

枚举类型的内部以字典方式实现，每个枚举值都有 name 和 value 属性。可通过名字或值查找对应枚举实例。

```
>>> X.B.name
'B'

>>> X.B.value
100
```

```
>>> X["B"]                          # by-name
<X.B: 100>

>>> X([1, 2, 3])                    # by-value
<X.C: [1, 2, 3]>
```

```
>>> X([1, 2])
ValueError: [1, 2] is not a valid X
```

按字典规则，值（value）可以相同，但名字（name）不许重复。

```
>>> class X(enum.Enum):
        A = 1
        B = 1
```

```
>>> class X(enum.Enum):
        A = 1
        A = 2

TypeError: Attempted to reuse key: 'A'
```

可当值相同时，无论基于名字还是值查找，都返回第一个定义项。

```
>>> class X(enum.Enum):
        A = 100
        B = "b"
        C = 100

>>> X.A
<X.A: 100>

>>> X.C
<X.A: 100>

>>> X["C"]
<X.A: 100>

>>> X(100)
<X.A: 100>
```

> 如要避免值相同的枚举定义，可用 enum.unique 装饰器。

相比于传统的枚举常量，标准库提供了丰富的扩展，包括自增数字、标志位等。只是这些需额外内存和性能开销，算是各有利弊。

内存

对于常用的小数字，解释器会在初始化时进行预缓存。稍后使用，直接将名字关联到这些缓存对象即可。如此一来，无须创建实例对象，可提高性能，且节约内存开销。

> 以 Python 3.6 为例，其预缓存范围是[-5, 256]。

具体过程请阅读 Objects/longobject.c : _PyLong_Init 源码。

```
>>> a = -5
>>> b = -5
>>> a is b
True

>>> a = 256
>>> b = 256
>>> a is b
True
```

如果超出缓存范围，那么每次都要新建对象，这其中包括内存分配等操作。

```
>>> a = -6
>>> b = -6
>>> a is b
False

>>> a = 257
>>> b = 257
>>> a is b
False
```

开发大数据等应用，免不了持有海量数字对象（比如超大 ID 列表）。Python 2 对回收后的整数复用内存不做收缩处理，会导致大量闲置内存驻留。而 Python 3 则改进了此设计，极大地减少了内存占用，这也算是迁移到新版的一个理由。

下面的测试代码在 Ubuntu 16.04 下分别以 Python 2.7 和 Python 3.6 运行，输出不同阶段的 RSS 内存大小。

示例中使用了 psutil 扩展库，且不同操作系统的测试结果可能有所不同。

test.py

```
from __future__ import print_function
import psutil
```

```
def rss():                                          # 输出进程 RSS 内存大小
    m = psutil.Process().memory_info()
    print(m.rss >> 20, "MB")


if __name__ == '__main__':
    rss()                                           # 起始内存占用
    x = list(range(10000000))                       # 使用列表持有海量数字对象
    rss()                                           # 输出海量数字造成的内存开销
    del x                                           # 删除列表，回收数字对象
    rss()                                           # 输出回收后的内存占用
```

输出：

```
$ python2.7 test.py
11  MB
323 MB
246 MB
```

```
$ python3.6 test.py
11  MB
397 MB
11  MB
```

2.2.2 浮点数

默认 float 类型存储双精度（double）浮点数，可表达 16 到 17 个小数位。

```
>>> 1/3
0.3333333333333333

>>> 0.1234567890123456789
0.12345678901234568
```

从实现方式看，浮点数以二进制存储十进制数的近似值。这可能导致执行结果与编码预期不符，造成不一致性缺陷。所以，对精度有严格要求的场合，应选择固定精度类型。

> 可通过 float.hex 方法输出实际存储值的十六进制格式字符串，以检查执行结果为何不同。

还可用该方式实现浮点值的精确传递，避免精度丢失。

```
>>> 0.1 * 3 == 0.3
False

>>> (0.1 * 3).hex()              # 显然两个存储内容并不相同
'0x1.3333333333334p-2'

>>> (0.3).hex()
'0x1.3333333333333p-2'
```

```
>>> s = (1/3).hex()

>>> float.fromhex(s)             # 反向转换回浮点数
0.3333333333333333
```

对于简单的比较操作，可尝试将浮点数限制在有效的固定精度内。但考虑到 round 算法实现问题，更准确的做法是用 decimal.Decimal 类型。

```
>>> round(0.1 * 3, 2) == round(0.3, 2)      # 避免不确定性，左右都使用固定精度
True

>>> round(0.1, 2) * 3 == round(0.3, 2)      # 将 round 返回值作为操作数，导致精度再次丢失
False
```

不同类型的数字之间，可直接进行加减和比较运算。

```
>>> 1.1 + 2
3.1

>>> 1.1 < 2
True

>>> 1.1 == 1
False
```

转换

将整数或字符串转换为浮点数很简单，且能自动处理字符串内的正负符号和空白符。只

是超出有效精度时，结果与字符串内容存在差异。

```
>>> float(100)
100.0

>>> float("-100.123")                    # 符号
-100.123

>>> float("\t 100.123\n")                # 空白符
100.123

>>> float("1.234E2")                     # 科学记数法
123.4
```

```
>>> float("0.1234567890123456789")
0.12345678901234568
```

反过来，将浮点数转换为整数时，有多种不同方案可供选择。可直接截掉小数部分，或分别往大小两个方向取临近整数。

```
>>> int(2.6), int(-2.6)                          # 截掉小数部分
(2, -2)
```

```
>>> from math import trunc, floor, ceil

>>> trunc(2.6), trunc(-2.6)                      # 截断小数部分
(2, -2)

>>> floor(2.6), floor(-2.6)                      # 往小数字方向取最近整数
(2, -3)

>>> ceil(2.6), ceil(-2.6)                        # 往大数字方向取最近整数
(3, -2)
```

十进制浮点数

与 float 基于硬件的二进制浮点类型相比，decimal.Decimal 是十进制实现，最高可提供 28 位有效精度。其能准确表达十进制数和运算，不存在二进制近似值问题。

```
>>> 1.1 + 2.2                              # 结果与 3.3 近似
3.3000000000000003

>>> (0.1 + 0.1 + 0.1 - 0.3) == 0           # 二进制近似值计算结果与十进制预期不符
False
```

```
>>> from decimal import Decimal

>>> Decimal("1.1") + Decimal("2.2")
Decimal('3.3')

>>> (Decimal("0.1") + Decimal("0.1") + Decimal("0.1") - Decimal("0.3")) == 0
True
```

在创建 Decimal 实例时，应该传入一个准确数值，比如整数或字符串等。如果是 float 类型，那么在构建之前，其精度就已丢失。

```
>>> Decimal(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')

>>> Decimal("0.1")
Decimal('0.1')
```

需要时，可通过上下文修改 Decimal 默认的 28 位精度。

```
>>> from decimal import Decimal, getcontext

>>> getcontext()
```

```
Context(prec=28, ...)

>>> getcontext().prec = 2

>>> Decimal(1) / Decimal(3)
Decimal('0.33')
```

或用 localcontext 限制某个区域的精度。

```
>>> from decimal import localcontext

>>> with localcontext() as ctx:                # 在该范围内将精度修改为 2，不会影响外部精度
        ctx.prec = 2
        print(getcontext().prec)
        print(Decimal(1) / Decimal(3))

2
0.33
```

> 除非有明确需求，否则不要用 Decimal 替代 float，要知道其运算速度会慢许多。

四舍五入

同样因为近似值和精度问题，造成对 float 进行“四舍五入”（round）操作存在不确定性，其结果会导致一些不易察觉的陷阱。

```
>>> round(0.5)          # 5 舍
0

>>> round(1.5)          # 5 入
2
```

按照 round 算法规则，按临近数字距离远近来考虑是否进位。如此，四舍六入就是确定的，相关问题都集中在两边距离相等的 5 是否进位。

> 以 0.4 为例，其舍入后的相邻数字是 0 和 1，从距离上看自然是 0 离 0.4 更近一些。

对于 5，还要考虑后面是否还有小数位。如果有，那么左右距离就不可能是相等的，这自然是要进位的。

```
>>> round(0.5)              # 与 0、1 距离相等，暂时不确定
0

>>> round(0.500001)         # 哪怕 0.5 后面的小数部分再小，那也表示它更接近 1
1

>>> round(1.25, 1)
1.2

>>> round(1.25001, 1)
1.3
```

剩下的，要看是返回整数还是浮点数。如果是整数，就取临近的偶数。

```
>>> round(0.5)              # 0 ---> 0.5 ---> 1
0

>>> round(1.5)              # 1 ---> 1.5 ---> 2
2

>>> round(2.5)              # 2 ---> 2.5 ---> 3
2
```

不同版本，规则存在差异。比如 Python 2.7，round(2.5)返回 3.0。

从这点来看，我们应谨慎对待此类行为差异，并严格测试其造成的影响。

如果依旧返回浮点数，事情就变得有点莫名其妙了。有些文章宣称“奇舍偶入”或“五成双”等，也就是看数字 5 前一位小数的奇偶来决定是否进位，但事实未必如此。

```
>>> round(1.25, 1)          # 偶舍
1.2

>>> round(1.245, 2)         # 偶进
1.25
```

```
>>> round(2.675, 2)          # 下面的都是奇数 7，但有舍有入
2.67

>>> round(2.375, 2)
2.38
```

对此，官方文档 *Floating Point Arithmetic: Issues and Limitations* 宣称这并非错误，而属于事出有因。对此，我们可改用 Decimal，按需求选取可控的进位方案。

```
>>> from decimal import Decimal, ROUND_HALF_UP

>>> def roundx(x, n):
        return Decimal(x).quantize(Decimal(n), ROUND_HALF_UP)      # 严格按照四舍五入进行
```

```
>>> roundx("1.24", ".1")
Decimal('1.2')

>>> roundx("1.25", ".1")
Decimal('1.3')

>>> roundx("1.26", ".1")
Decimal('1.3')
```

```
>>> roundx("1.245", ".01")
Decimal('1.25')

>>> roundx("2.675", ".01")
Decimal('2.68')

>>> roundx("2.375", ".01")
Decimal('2.38')
```

2.2.3 字符串

字符串存储 Unicode 文本，是不可变序列类型。相比于 Python 2 中的混乱，Python 3 总算顺应了时代的发展，将文本和二进制彻底分离。

Unicode 的设计意图是为了解决跨语言、跨平台转换和处理需求，其用统一编码方案容纳不同国家地区的文字，以解决传统编码方案的不兼容问题，故又称作统一码、万国码。

Unicode 为每个字符分配一个被称作码点（code point）的整数序号，对应的编码方案叫作通用字符集（Universal Character Set, UCS）。依编码整数长度，其又分为 UCS-2 和 UCS-4 两种，后者为 32 位，可容纳更多字符。UCS 只规定了字符和码点的对应关系，并不涉及如何显示和存储。

UTF（Unicode Transformation Format）的作用是将码点整数转换为计算机可存储的字节格式。发展至今，有 UTF-8、UTF-16、UTF-32 等多种方案。其中 UTF-8 采用变长格式，因其与 ASCII 兼容，故此是当下使用最广泛的一种。对于以英文为主的内容，UTF-8 可获得最好的存储效率。而使用两字节等长方案的 UTF-16，有更快的处理效率，常被用作执行编码。

UTF 还可在文本头部插入被称作 BOM（byte order mark）的标志来标明字节序信息，以区分大小端（BE、LE）。因此，其又可细分为 UTF-16LE、UTF-32BE 等。

```
>>> s = "汉字"

>>> len(s)
2
```

```
>>> hex(ord("汉"))              # code point
0x6c49

>>> chr(0x6c49)
汉
```

```
>>> ascii("汉字")               # 对 non-ASCII 进行转义
\u6c49\u5b57
```

字符串字面量（literal）以成对的单引号、双引号，或可跨行的三引号语法构成，自动合并相邻字面量。其支持转义、八进制、十六进制，或 Unicode 格式字符。

用单引号还是双引号，并没什么特殊限制。如果文本内的引用文字使用双引号，那么外面用单引号可避免转义，更易阅读。通常情况下，建议遵循多数编程语言惯例，使用双引号标示。除了单引号在英文句法里的特殊用途外，它还常用来表示单个字符。

```
>>> "h\x69, \u6C49\U00005B57"
hi, 汉字
```

注意：Unicode 格式大小写分别表示 16 位（\u）和 32 位（\U）整数，不能混用。

```
>>> "It's my life"                                    # 英文缩写
>>> 'The report contained the "facts" of the case.'   # 包含引文，避免使用\"转义
```

```
>>> "hello" ", " "world"                              # 自动合并多个相邻的字面量
hello, world
```

```
>>> """                                               # 换行符、前导空格、空行都是组成内容
    Beautiful is better than ugly.
    Explicit is better than implicit.
    Simple is better than complex.
"""
```

在字面量前添加标志，指示构建特定格式字符串。

最常用的原始字符串（r, raw string），它将反斜线视作字符内容，而非转义标志。这在构建类似 Windows 路径、正则表达式匹配模式之类的文法字符串时很有用。

```
>>> open(r"c:\windows\readme.txt")          # Windows 路径

>>> re.findall(r"\b\d+\b", "a10 100")       # 正则表达式
['100']
```

```
>>> type(u"abc")                            # 默认 str 就是 unicode，无须添加 u 前缀
str

>>> type(b"abc")                            # 构建字节数组
bytes
```

操作

支持用加法或乘法运算符拼接字符串。

```
>>> s = "hello"
>>> s += ", world"

>>> "-" * 10
----------
```

编译器会尝试在编译期直接计算出字面量拼接结果，可避免运行时开销。不过此类优化程度有限，并不总是有效。

```
>>> def test():
        a = "x" + "y" + "z"
        b = "a" * 10
        return a, b


>>> dis.dis(test)
  2           0 LOAD_CONST               7 ('xyz')             # 直接给出结果，省略加法运算
  3           4 LOAD_CONST               8 ('aaaaaaaaaa')      # 省略乘法运算
```

至于多个动态字符串拼接，应优先选择 join 或 format 方式。

> 相比于多次加法运算和多次内存分配（字符串是不可变对象），join 这类函数（方法）可预先计算出总长度，一次性分配内存，随后直接复制内存数据填充。另一方面，将固定模板内容与变量分离的 format，更易阅读和维护。

```
>>> username = "qyuhen"
>>> datetime = "2017010"

>>> "/data/" + username + "/message/" + datetime + ".txt"         # 糟糕的方式
/data/qyuhen/message/20170101.txt
```

```
>>> tmpl = "/data/{user}/message/{time}.txt"                      # 模板单独维护
>>> tmpl.format(user = username, time = datetime)                 # 无须考虑参数的次序
/data/qyuhen/message/20170101.txt
```

让我们对比用加法和 join 拼接 26 个大写字母的性能差异。虽然此测试不具备代表性，但可以提供一个粗略的验证方法。

test.py

```
import string
x = list(string.ascii_uppercase)


@profile
def test_add():
    s = ""
    for c in x:
        s += c
    return s


@profile
def test_join():
    return "".join(x)

test_add()
test_join()
```

输出：

```
$ kernprof -l -v test.py

Timer unit: 1e-06 s

Total time: 2.5e-05 s
Function: test_add
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     5                                           @profile
     6                                           def test_add():
     7         1            4      4.0     16.0      s = ""
     8        27            9      0.3     36.0      for c in x:
     9        26           12      0.5     48.0          s += c
    10         1            0      0.0      0.0      return s


Total time: 1e-06 s
Function: test_join
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
```

```
    13                                          @profile
    14                                          def test_join():
    15         1            1      1.0    100.0      return "".join(x)
```

> 有关 line_profiler、memory_profiler 的使用方法，请参阅第 11 章。

编写代码时除保持简单外，还应具备良好的可读性。比如判断是否包含子串，in、not in 操作符就比 find 方法自然，更贴近日常阅读习惯。

```
>>> "py" in "python"
True

>>> "Py" not in "python"
True
```

作为序列类型，可使用索引序号访问字符串的单个字符或者某一片段。Python 3 支持负索引，也就是反向从尾部以−1 开始（索引 0 表示正向第一个字符）。

```
>>> s = "0123456789"

>>> s[2]
'2'

>>> s[-1]
'9'

>>> s[2:6]
'2345'

>>> s[2:-2]
'234567'
```

> 使用两个索引号表示一个序列片段的语法被称作切片（slice），可以此返回字符串子串。
>
> 无论以哪种方式返回与原字符串内容不同的子串时，都可能会重新分配内存，并复制数据。

先看相同或不同内容时，字符串对象的构建情形。

```
>>> s = "-" * 1024
>>> s1 = s[10:100]                # 片段，内容不同
>>> s2 = s[:]                     # 内容相同
>>> s3 = s.split(",")[0]          # 内容相同

>>> s1 is s                       # 内容不同，构建新对象
False

>>> s2 is s                       # 内容相同时，直接引用原字符串对象
True

>>> s3 is s
True
```

进一步观察内存分配状况。

test.py

```
@profile
def test():
    a = x[10:-10]               # 截取子串
    b = x.split(",")            # 分割字符串
    return a, b


x = "0," * (1 << 20)
test()
```

输出：

```
$ python -m memory_profiler test.py

Line #      Mem usage      Increment     Line Contents
================================================
     1     39.574 MiB      0.000 MiB     @profile
     2                                   def test():
     3     41.582 MiB      2.008 MiB         a = x[10:-10]
     4     49.789 MiB      8.207 MiB         b = x.split(",")
     5     49.789 MiB      0.000 MiB         return a, b
```

> 内存分配行为，与具体 Python 实现版本有关，不能一概而论。

字符串类型包含丰富的处理方法，可满足大多数操作的需要。对于更复杂的文本处理，还可使用正则表达式或专业扩展库，比如 NLTK、TextBlob 等。

转换

除了与数字、Unicode 码点的转换外，最常见的是在不同编码间进行转换。

> Python 3 使用 bytes、bytearray 存储字节序列，不再和 str 混用。

```
>>> s = "汉字"

>>> b = s.encode("utf-16")                # to bytes
>>> b.decode("utf-16")                    # to Unicode str
'汉字'
```

如要处理 BOM 信息，可导入 codecs 模块。

```
>>> s = "汉字"

>>> s.encode("utf-16").hex()
'fffe496c575b'

>>> codecs.BOM_UTF16_LE.hex()               # BOM 标志
'fffe'
```

```
>>> codecs.encode(s, "utf-16be").hex()       # 按指定 BOM 转换
'6c495b57'

>>> codecs.encode(s, "utf-16le").hex()
'496c575b'
```

另外，Python 3 默认编码不再是 ASCII，无须额外设置。

```
>>> sys.getdefaultencoding()
'utf-8'
```

格式化

长期发展下来，Python 累积了多种字符串格式化方式。与古老的面孔相比，人们更喜欢或倾向于使用新的特征。

Python 3.6 新增了 f-strings 支持，这在多数脚本语言里属于标配。

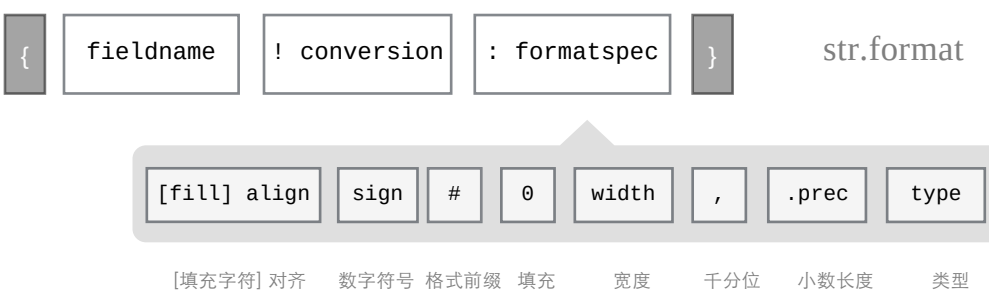
> 使用 f 前缀标志，解释器解析大括号内的字段或表达式，在上下文名字空间查找同名对象进行值替换。格式化控制依旧遵循 format 规范，但阅读体验上却更加完整和简洁。

```
>>> x = 10
>>> y = 20

>>> f"{x} + {y} = {x + y}"                      # f-strings
10 + 20 = 30

>>> "{} + {} = {}".format(x, y , x + y)         # str.format
10 + 20 = 30
```

除运算符外，还可以是函数调用。

```
>>> f"{type(x)}"
"<class 'int'>"
```

完整的 format 格式化以位置序号或字段名匹配参数进行值替换，可添加对齐、填充、精度等控制。从某种角度看，f-strings 有点像 format 的增强语法糖。



将两者进行对比，f-strings 类模板方式更加灵活，其一定程度上将输出样式与数据来源分离。但其缺点是与上下文名字耦合，导致模板内容与代码必须保持同步修改。而 format 的序号与主键匹配方式可避开这一点，且支持静态分析工具检查，只可惜它不支持表达式。

手工序号和自动序号

```
>>> "{0} {1} {0}".format("a", 10)
a 10 a

>>> "{} {}".format(1, 2)                          # 自动序号，不能与手工序号混用
1 2
```

主键

```
>>> "{x} {y}".format(x = 100, y = [1,2,3])
100 [1, 2, 3]
```

属性和索引

```
>>> x.name = "jack"

>>> "{0.name}".format(x)                          # 对象属性
jack

>>> "{0[2]}".format([1,2,3,4])                    # 索引
3
```

宽度、补位

```
>>> "{0:#08b}".format(5)
0b000101
```

数字

```
>>> "{:06.2f}".format(1.234)                      # 保留 2 位小数
001.23

>>> "{:,}".format(123456789)                      # 千分位
123,456,789
```

对齐

```
>>> "[{:^10}]".format("abc")                      # 居中
```

```
[   abc    ]

>>> "[{:.<10}]".format("abc")                    # 左对齐，以点填充
[abc.......]
```

> 古老的 printf 百分号格式化方式已被官方标记为“obsolete”，加上其自身固有的一些问题，可能会被后续版本抛弃，在此不建议使用。另外，标准库里的 string.Template 功能弱，且性能也差，同样不建议使用。

池化

字符串可能是进程里实例数量最多的类型之一，因为无处不在的名字就是字符串实例。

鉴于相同名字会重复出现在不同的名字空间里，那么有必要共享实例。内容相同，且不可变，共享不会导致任何问题。关键是节约内存，且可省去创建新实例的开销。

对此，Python 的做法是实现一个字符串池（intern）。

> 池负责管理实例，使用者只需引用即可。另一潜在的好处是，从池返回的字符串，只需比较指针就可知道内容是否相同，无须额外计算。可以用池来提升哈希表等类似结构的查找性能。

```
>>> "__name__" is sys.intern("__name__")
True
```

除以常量方式出现的名字和字面量外，动态生成的字符串一样可加入池中。如此可保证每次都引用同一对象，不会有额外的创建和分配操作。

```
>>> a = "hello, world!"
>>> b = "hello, world!"

>>> a is b                                       # 不同实例
False
>>> sys.intern(a) is sys.intern("hello, world!")  # 相同实例
True
```

当然，一旦失去所有外部引用，池内的字符串对象一样会被回收。

```
>>> a = sys.intern("hello, world!")

>>> id(a)
4401879024

>>> id(sys.intern("hello, world!"))        # 有外部引用
4401879024
```

```
>>> del a                                  # 删除外部引用后被回收

>>> id(sys.intern("hello, world!"))        # 从 id 值不同可以看到字符串是新建后入池的
4405219056
```

字符串池的实现算法很简单，就是简单的字典结构。

详情可参考 Objects/unicodeobject.c : PyUnicode_InternInPlace。

做大数据处理时，可能须创建海量主键，使用类似机制有助于减少对象数量，节约大量内存。当然，可以选择更高效的数据结构，而不一定是系统内置的字符串池。

2.2.4 字节数组

虽然生物都由细胞构成，但在普通人的眼里，并不会将人、狮子、花草这些当作细胞看待。因为对待事物的角度决定了，我们更关心生物的外在形状和行为，而不是其组织构成。

从底层实现来说，所有数据都是二进制字节序列。但为了更好地表达某个逻辑，我们将其抽象成不同类型，一如细胞和狮子的关系。当谈及字节序列时，我们更关心的是存储和传输方式；而面向类型时，则着重于其抽象属性。尽管一体两面，但从不混为一谈。

因此，当 str 瘦身只为字符串而存在时，专门用于二进制数据处理的类型也必然会出现。

早在 Python 2.6 时就引入了 bytearray 字节数组，后 Python 3 又新增了只读版本的 bytes。

同为不可变序列类型，bytes 与 str 有着非常类似的操作方式。

```
>>> b"abc"
>>> bytes("汉字", "utf-8")
```

```
>>> a = b"abc"
>>> b = a + b"def"

>>> b
b'abcdef'

>>> b.startswith(b"abc")
True

>>> b.upper()
b'ABCDEF'
```

相比于 bytes 的一次性内存分配，bytearray 可按需扩张，更适合作为可读写缓冲区使用。如有必要，还可为其提前分配足够的内存，避免中途扩张造成额外消耗。

```
>>> b = bytearray(b"ab")

>>> len(b)
2

>>> b.append(ord("c"))
>>> b.extend(b"de")

>>> b
bytearray(b'abcde')
```

其同样支持加法、乘法等运算符。

```
>>> b"abc" + b"123"
b'abc123'

>>> b"abc" * 2
b'abcabc'
```

```
>>> a = bytearray(b"abc")
```

```
>>> a * 2
bytearray(b'abcabc')

>>> a += b"123"

>>> a
bytearray(b'abc123')
```

内存视图

如果要引用字节数据的某个片段，该怎么做？需要考虑的问题如下：是否会有数据复制行为？是否能同步修改？

```
>>> a = bytearray([0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16])

>>> x = a[2:5]                                    # 引用片段?
>>> x
bytearray(b'\x12\x13\x14')
```

```
>>> a[3] = 0xEE                                   # 修改原数据
>>> a
bytearray(b'\x10\x11\x12\xee\x14\x15\x16')

>>> x                                             # 并未同步发生变更，显然是数据复制
bytearray(b'\x12\x13\x14')
```

为什么需要引用某个片段，而不是整个对象？

以自定义网络协议为例，通常由标准头和数据体两部分组成。如要验证数据是否被修改，总不能将整个包作为参数交给验证函数。这势必要求该函数了解协议包结构，这显然是不合理的设计。而复制数据体又可能导致重大性能开销，同样得不偿失。

鉴于 Python 没有指针概念，外加内存安全模型的限制，要做到这一点似乎并不容易。因此，须借助一种名为内存视图（Memory Views）的方式来访问底层内存数据。

内存视图要求目标对象支持缓冲协议（Buffer Protocol）。它直接引用目标内存，没有额外复制行为。因此，可读取最新数据。在目标对象允许的情况下，还可执行写操作。

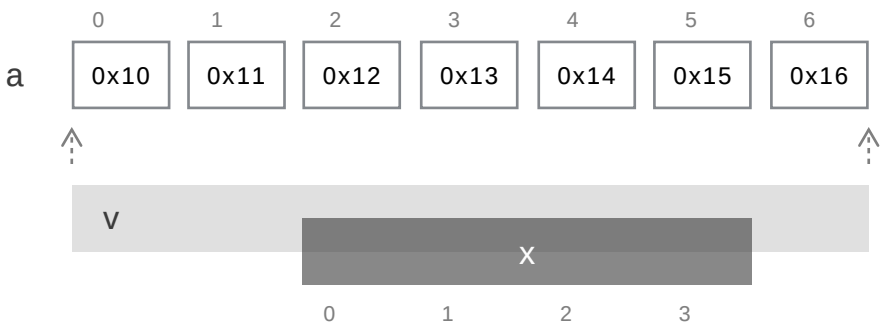
常见支持视图操作的有 bytes、bytearray、array.array，以及著名扩展库 NumPy 的某些类型。

```
>>> a = bytearray([0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16])

>>> v = memoryview(a)                    # 完整视图
```

```
>>> x = v[2:5]                           # 视图片段
>>> x.hex()
'121314'
```

```
>>> a[3] = 0xee                          # 对原数据修改，可通过视图观察到

>>> x.hex()
'12ee14'
```

```
>>> x[1] = 0x13                          # 因引用相同的内存区域，可通过视图修改原数据

>>> a
bytearray(b'\x10\x11\x12\x13\x14\x15\x16')
```



视图片段有自己的索引范围。读写操作以视图索引为准，但不得超出限制。

当然，能否通过视图修改数据，须看原对象是否允许。

```
>>> a = b"\x10\x11"                          # bytes 是不可变类型
>>> v = memoryview(a)

>>> v[1] = 0xEE
TypeError: cannot modify read-only memory
```

如要复制视图数据，可调用 tobytes、tolist 方法。复制后的数据与原对象无关，同样不会影响视图自身。

```
>>> a = bytearray([0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16])
>>> v = memoryview(a)
>>> x = v[2:5]
```

```
>>> b = x.tobytes()                # 复制并返回视图数据
>>> b
b'\x12\x13\x14'
```

```
>>> a[3] = 0xEE                    # 对原数据进行修改

>>> b                              # 不影响复制数据
b'\x12\x13\x14'
```

除了上述这些，内存视图还为我们提供了一种内存管理手段。

> 比如，通过 bytearray 预申请很大的一块内存，随后以视图方式将不同片段交由不同逻辑使用。因逻辑不能越界访问，故此可实现简易的内存分配器模式。对于 Python 这种限制较多的语言，合理使用视图可在不使用 ctypes 等复杂扩展的前提下，改善算法性能。
>
> 可使用 memoryview.cast、struct.unpack 将字节数组转换为目标类型。

2.2.5 列表

仅从操作方式上看，列表像是数组和链表的综合体。除按索引访问外，还支持插入、追加、删除等操作，完全可当作队列或栈使用。如不考虑性能问题，列表似乎是一种易用且功能完善的理想数据结构。

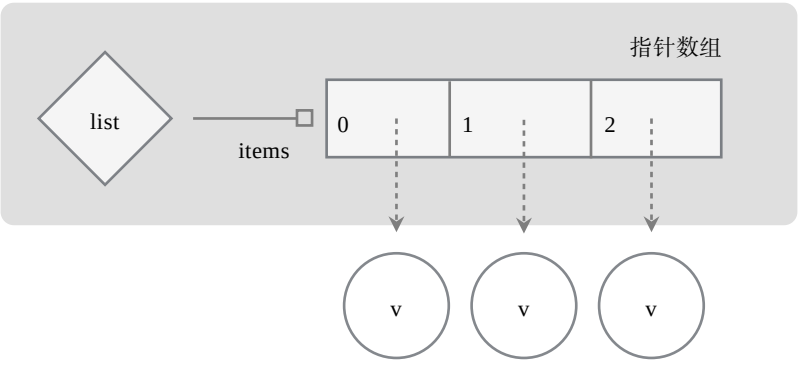
```
>>> x = [1, 2]
>>> x[1]
2

>>> x.insert(0, 0)
>>> x
[0, 1, 2]

>>> x.reverse()
>>> x
[2, 1, 0]
```

queue

```
>>> q = []

>>> q.append(1)                    # 向队列追加数据
>>> q.append(2)

>>> q.pop(0)                       # 按追加顺序弹出数据
1

>>> q.pop(0)
2
```

> 如有大量写操作，建议使用 collections.deque、queue 等类型。

列表的内部结构由两部分组成，保存元素数量和内存分配计数的头部，以及存储元素指针的独立数组。所有元素项使用该数组保存指针引用，并不嵌入实际内容。



作为使用频率最高的数据结构之一，列表的性能优化很重要。固定长度的头部结构，很容易实现内存复用。创建时，优先从复用区获取。被回收时，除非超出复用数量限制（80），否则会被放回复用区，而非交还内存。每次真正需要分配和释放的是指针数组。

用数组而非链表存储元素项引用，也有实际考虑。从读操作来看，无论遍历还是基于序号访问，数组性能总是最高的。尽管插入、删除等变更操作须移动内存，但也仅仅是指针复制，无关元素大小，不会有太高消耗。如果列表太大，或写操作远多于读，那么应使用针对性的数据结构，而非通用设计的内置列表类型。

另外，指针数组内存分配算法基于元素数量和剩余空间大小，按相应比率进行扩容或收缩，非逐项进行。如此，可避免太频繁的内存分配操作。

构建

显式指定元素的构建语法最为常用。也可基于类型创建实例，接收可迭代对象作为初始内容。不同于数组，列表仅存储指针，而对元素内容并不关心，故可以是不同类型混合。

```
>>> [1, "abc", 3.14]
[1, 'abc', 3.14]
```

```
>>> list("abc")                    # iterable
['a', 'b', 'c']

>>> list(range(3))
[0, 1, 2]
```

另有被称作推导式（comprehension）的语法。同样使用方括号，但以 for 循环初始化元素，并可选 if 表达式作为过滤条件。

```
>>> [x + 1 for x in range(3)]
[1, 2, 3]

>>> [x for x in range(6) if x % 2 == 0]
[0, 2, 4]
```

其行为类似于以下代码。

```
>>> d = []

>>> for x in range(6):
        if x % 2 == 0:
            d.append(x)

>>> d
[0, 2, 4]
```

专有名词 Pythonic 的基本意思就是写出简洁优雅的 Python 代码，推导式算其中的一种。

无论是历史原因，还是实现方式，内置类型关注性能要多过设计。如要实现自定义列表，建议基于 collections.UserList 包装类型完成。除统一 collections.abc 体系外，最重要的是该类型重载并完善了相关运算符方法。

```
>>> list.__bases__
(object,)

>>> collections.UserList.__bases__
(collections.abc.MutableSequence,)
```

以加法运算符为例，对比不同继承的结果。

```
>>> class A(list): pass

>>> type(A("abc") + list("de"))              # 返回的是 list，而不是 A
list
```

```
>>> class B(collections.UserList): pass

>>> type(B("abc") + list("de"))              # 返回 B 类型
__main__.B
```

最小接口设计是个基本原则，应慎重考虑列表这种功能丰富的类型是否适合作为基类。

操作

用加法运算符连接多个列表，乘法复制内容。

```
>>> [1, 2] + [3, 4]
[1, 2, 3, 4]

>>> [1, 2] * 2
[1, 2, 1, 2]
```

注意，同是加法（或乘法）运算，但结果却不同。

```
>>> a = [1, 2]
>>> b = a

>>> a = a + [3, 4]                  # 新建列表对象，然后与 a 关联

>>> a                               # a、b 结果不同，确定 a 指向新对象
[1, 2, 3, 4]

>>> b
[1, 2]
```

```
>>> a = [1, 2]
>>> b = a

>>> a += [3, 4]                     # 直接修改 a 内容

>>> a                               # a、b 结果相同，确认修改原对象
[1, 2, 3, 4]

>>> b
[1, 2, 3, 4]

>>> a is b
True
```

> 编译器将“+=”运算符处理成 INPLACE_ADD 操作，也就是修改原数据，而非新建对象。其效果类似于执行 list.extend 方法。

判断元素是否存在，习惯使用 in，而非 index 方法。

```
>>> 2 in [1, 2]
True
```

至于删除操作，可用索引序号指定单个元素，或切片指定删除范围。

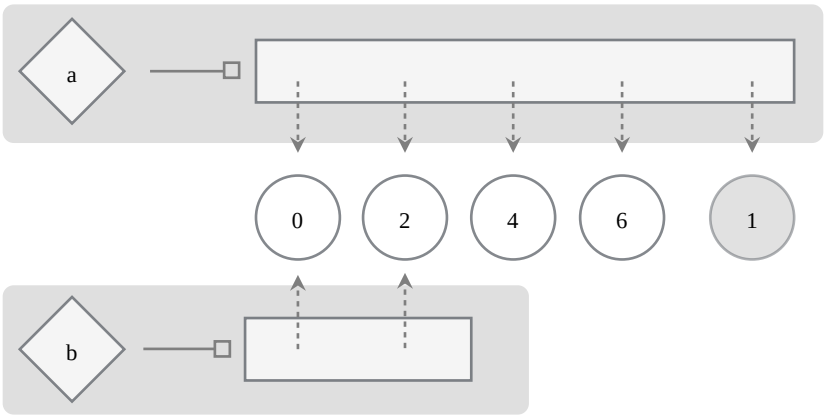
```
>>> a = [0, 1, 2, 3, 4, 5]

>>> del a[5]                          # 删除单个元素

>>> a
[0, 1, 2, 3, 4]

>>> del a[1:3]                        # 删除范围

>>> a
[0, 3, 4]
```

返回切片时创建新列表对象，并复制相关指针数据到新数组。除所引用目标相同外，对列表自身的修改（插入、删除等）互不影响。

```
>>> a = [0, 2, 4, 6]
>>> b = a[:2]

>>> a[0] is b[0]                  # 复制引用，依然指向同一对象
True

>>> a.insert(1, 1)                # 对 a 列表的操作，不会影响 b

>>> a
[0, 1, 2, 4, 6]

>>> b
[0, 2]
```

复制的是指针（引用），而非目标元素对象。

对列表自身的修改互不影响，但对目标元素对象的修改是共享的。



列表排序可设定条件，比如按字段或长度等。

```
class User:

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __repr__(self):
        return f"{self.name} {self.age}"
```

```
>>> users = [User(f"user{i}", i) for i in (3, 1, 0, 2)]

>>> users
[user3 3, user1 1, user0 0, user2 2]
```

```
>>> users.sort(key = lambda u: u.age)          # 使用 lambda 匿名函数返回排序条件

>>> users
[user0 0, user1 1, user2 2, user3 3]
```

如要返回复制品，可使用 sorted 函数。

```
>>> d = [3, 0, 2, 1]

>>> sorted(d)                                  # 同样可指定排序条件，或倒序
[0, 1, 2, 3]
```

```
>>> d                                          # 并未影响原列表
[3, 0, 2, 1]
```

利用 bisect 模块，可向有序列表插入元素。它使用二分法查找适合位置，可用来实现优先级队列或一致性哈希算法。

```
>>> d = [0, 2, 4]

>>> bisect.insort_left(d, 1)                  # 插入新元素后，依然保持有序状态
>>> d
[0, 1, 2, 4]

>>> bisect.insort_left(d, 2)
>>> d
[0, 1, 2, 2, 4]

>>> bisect.insort_left(d, 3)
>>> d
[0, 1, 2, 2, 3, 4]
```

> 自定义复合类型，可通过重载比较运算符（__eq__、__lt__ 等）实现自定义排序条件。

元组

尽管两者并没有直接关系，但在操作方式上，元组可被当作列表的只读版本使用。

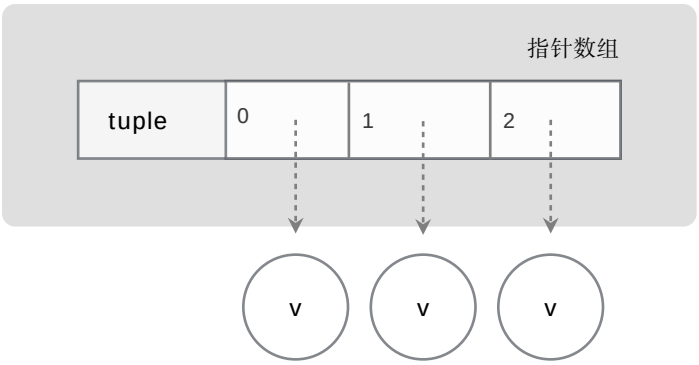
```
>>> a = tuple([1, "abc"])
>>> a[0] = 100
TypeError: 'tuple' object does not support item assignment
```

元组使用小括号语法，但要与普通括号区别开来。

```
>>> a = (1, )               # 仅一个元素的元组
>>> type(a)
tuple

>>> b = (1)                 # 普通括号
>>> type(b)
int
```

因元组是不可变类型，它的指针数组无须变动，故一次性完成内存分配。系统会缓存复用一定长度的元组内存（含指针数组）。创建时，按所需长度提取复用，没有额外内存分配。从这点上来看，元组的性能要好于列表。

Python 3.6 缓存复用长度在 20 以内的 tuple 内存，每种 2000 上限。



IPython

```
>>> %timeit [1, 2, 3]
64.8 ns ± 0.375 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

```
>>> %timeit (1, 2, 3)
16.4 ns ± 0.156 ns per loop (mean ± std. dev. of 7 runs, 100000000 loops each)
```

支持与列表类似的运算符操作。但不能修改，总是返回新对象。

```
>>> (1, 2) + (3, 4)
(1, 2, 3, 4)

>>> (1, 2) * 2
(1, 2, 1, 2)
```

```
>>> a = (1, 2, 3)
>>> b = a

>>> a += (4, 5)                    # 创建新 tuple，而不是修改原内容

>>> a
```

```
(1, 2, 3, 4, 5)

>>> b
(1, 2, 3)
```

因列表支持插入、删除等修改操作，所以序号无法与元素对象构成固定映射。但元组不同，相同序号总是返回同一对象，故可为序号取个“别名”。

```
>>> User = collections.namedtuple("User", "name,age")      # 创建 User 类型，指定字段

>>> issubclass(User, tuple)                                # tuple 子类
True
```

```
>>> u = User("qyuhen", 60)

>>> u.name, u.age                       # 使用字段名访问
('qyuhen', 60)

>>> u[0] is u.name                      # 或依旧使用序号
True
```

> 对于定义纯数据类型，显然 namedtuple 要比 class 简洁。关键在于，名字要比序号更易阅读和更易维护，其类似于数字常量定义。

数组

数组与列表、元组的本质区别在于：元素单一类型和内容嵌入。

```
>>> import array

>>> a = array.array("b", [0x11, 0x22, 0x33, 0x44])

>>> memoryview(a).hex()                        # 使用内存视图查看，内容嵌入而非指针
11223344
```

```
>>> a = array.array("i")
>>> a.append(100)

>>> a.append(1.23)
```

```
TypeError: integer argument expected, got float
```

可直接存储包括 Unicode 字符在内的各种数字。

复合类型须用 struct、marshal、pickle 等转换为二进制字节后再行存储。

与列表类似，数组长度不固定，按需扩张或收缩内存。

```
>>> a = array.array("i", [1, 2, 3])

>>> a.buffer_info()                      # 返回缓冲区的内存地址和长度
(4481545888, 3)

>>> a.extend(range(100000))              # 追加大量内容后，内存地址和长度发生变化

>>> a.buffer_info()
(4460855296, 100003)
```

由于可指定更紧凑的数字类型，故数组可节约更多内存。再者，内容嵌入也避免了对象的额外开销，减少了活跃对象的数量和内存分配的次数。

test.py

```
import array


@profile
def test_list():
    x = []
    x.extend(range(1000000))
    return x


@profile
def test_array():
    x = array.array("l")
    x.extend(range(1000000))
    return x
```

```
test_array()
test_list()
```

输出：

```
$ python -m memory_profiler test.py

Line #      Mem usage      Increment     Line Contents
================================================
     4     45.699 MiB      0.000 MiB     @profile
     5                                   def test_list():
     6     45.699 MiB      0.000 MiB         x = []
     7     84.277 MiB     38.578 MiB         x.extend(range(1000000))
     8     84.277 MiB      0.000 MiB         return x


Line #      Mem usage      Increment     Line Contents
================================================
    11     37.723 MiB      0.000 MiB     @profile
    12                                   def test_array():
    13     37.727 MiB      0.004 MiB         x = array.array("l")
    14     45.699 MiB      7.973 MiB         x.extend(range(1000000))
    15     45.699 MiB      0.000 MiB         return x
```

2.2.6 字典

字典是内置类型中唯一的映射（mapping）结构，基于哈希表存储键值对数据。

值可以是任意数据，但主键必须是可哈希类型。常见的可变类型，如列表、集合等都不能作为主键使用。即便是元组等不可变类型，也不能引用可变类型元素。

```
>>> issubclass(list, collections.Hashable)
False

>>> issubclass(int, collections.Hashable)
True
```

```
>>> hash((1, 2, 3))
2528502973977326415
```

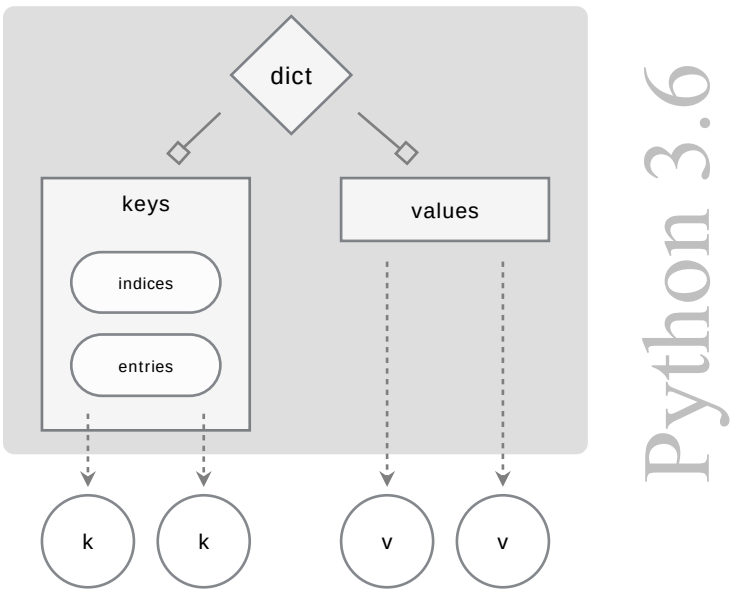
```
>>> hash((1, 2, [3, 4]))                    # 包含可变列表元素
TypeError: unhashable type: 'list'
```

哈希计算通过调用__hash__方法返回整数值，用来快速比较内容是否相同。某些类型虽然有该方法，但实际无法执行，故不能作为主键使用。另外，主键对象哈希值须恒定不变，否则无法查找键值，甚至引发错误。

```
>>> callable(list().__hash__)
False
```

自定义类型默认实现了__hash__和__eq__方法，用于哈希和相等比较操作。前者为每个实例返回随机值；后者除非与自己比较，否则总是返回 False。这里可根据需要重载。

作为常用数据结构，又因名字空间的缘故，字典的使用频率非常高。开发团队也一直致力于改进其数据结构和算法，这其中自然也包括惯用的缓存复用。



Python 3.6 借鉴 PyPy 字典设计，采用更紧凑的存储结构。keys.entries 和 values 用数组按添加顺序存储主键和值引用。实际哈希表由 keys.indices 数组承担，通过计算主键哈希值找到合适

的位置，然后在此存储主键在 keys.entries 的实际索引。如此一来，只要通过 indices 获取实际索引后，就可读取主键和值信息了。

虽然该版本按添加顺序存储，但内部实现不能作为依赖条件。在后续版本中，可能有其他变化。如有明确顺序需求，建议使用 collections.OrderedDict。

系统分别缓存复用 80 个 dict 和 keys，其中包括长度为 8 的 entries 内存。对于大量小字典对象而言，直接使用，无须任何内存分配操作。回收时，凡内存被扩张过的都被放弃。

从开放地址法（open-address）实现方式来看，它并不适合处理大数据。轻量级方案可选用 shelve、dbm 等标准库模块，也可直接采用 SQLite、LevelDB 等专业数据库。

构建

以大括号键值对方式创建，或调用类型构造。

```
>>> {"a": 1, "b": 2}
{'a': 1, 'b': 2}

>>> dict(a = 1, b = 2)
{'a': 1, 'b': 2}
```

初始化键值参数也可以元组、列表等可迭代对象方式提供。

```
>>> kvs = (("a", 1), ["b", 2])

>>> dict(kvs)
{'a': 1, 'b': 2}
```

基于动态数据创建时，多以 zip、map 函数或推导式完成。

```
>>> dict(zip("abc", range(3)))
{'a': 0, 'b': 1, 'c': 2}

>>> dict(map(lambda k, v: (k, v + 10), "abc", range(3)))      # 使用 lambda 匿名函数过滤数据
{'a': 10, 'b': 11, 'c': 12}

>>> {k: v + 10 for k, v in zip("abc", range(3))}              # 使用推导式处理数据
{'a': 10, 'b': 11, 'c': 12}
```

除直接提供内容外，某些时候，还须根据一定条件初始化字典对象。比如，基于已有字典内容扩展，或者初始化零值等。

```
>>> a = {"a": 1}

>>> b = dict(a, b = 2)                                  # 在复制 a 内容的基础上，新增键值对
>>> b
{'a': 1, 'b': 2}
```

```
>>> c = dict.fromkeys(b, 0)                             # 仅用 b 主键，内容另设
>>> c
{'a': 0, 'b': 0}
```

```
>>> d = dict.fromkeys(("counter1", "counter2"), 0)      # 显式提供主键
>>> d
{'counter1': 0, 'counter2': 0}
```

> 相比于 fromkeys 方法，推导式可完成更复杂的操作，比如额外的 if 过滤条件。

操作

字典不是序列类型，不支持序号访问，以主键读取、新增或删除内容。

```
>>> x = dict(a = 1)
>>> x["a"]                          # 读取
1

>>> x["b"] = 2                      # 修改或新增
>>> x
{'a': 1, 'b': 2}

>>> del x["a"]                      # 删除
>>> x
{'b': 2}
```

如主键不存在，会引发异常。可先以 in、not in 语句判断，或用 get 方法返回默认值。

```
>>> x = dict(a = 1)
```

```
>>> x["b"]
KeyError: 'b'

>>> "b" in x
False
```

```
>>> x.get("b", 100)              # 主键 b 不存在，返回默认值 100
100

>>> x.get("a", 100)              # 主键 a 存在，返回实际内容
1
```

> 方法 get 的默认值参数仅返回，不影响字典内容。但某些时候，我们还须向字典插入默认值。比如用字典存储多个计数器，那么在第一次取值时延迟初始化很有必要。在字典内有零值内容代表该计数曾被使用，没有则无法记录该行为。

```
>>> x = {}
>>> x.setdefault("a", 0)          # 如果有 a，那么返回实际内容，否则新增{a:0}键值
0

>>> x
{'a': 0}
```

```
>>> x["a"] = 100

>>> x.setdefault("a", 0)
100
```

字典不支持加法、乘法、大小等运算，但可比较内容是否相同。

```
>>> {"b":2, "a":1} == {"a":1, "b":2}
True
```

视图

与早期版本复制数据并返回列表不同，Python 3 默认以视图关联字典内容。如此一来，既能避免复制开销，还能同步观察字典变化。

```
>>> x = dict(a = 1, b = 2)
>>> ks = x.keys()                              # 主键视图

>>> "b" in ks                                  # 判断主键是否存在
True

>>> for k in ks: print(k, x[k])                # 利用视图迭代字典
a 1
b 2
```

```
>>> x["b"] = 200                               # 修改字典内容
>>> x["c"] = 3

>>> for k in ks: print(k, x[k])                # 视图能同步变化
a 1
b 200
c 3
```

字典没有独立的只读版本，无论传递引用还是复制品，都存在弊端。直接引用有被接收方修改内容的风险，而复制品又仅是一次性快照，无法获知字典变化。视图则不同，它能同步读取字典内容，却无法修改。且可选择不同粒度的内容进行传递，如此可将接收方限定为指定模式下的观察员。

```
def test(d):                        # 传递键值视图（items），只能读取，无法修改
    for k, v in d:
        print(k, v)
```

```
>>> x = dict(a = 1)
>>> d = x.items()

>>> test(d)
a 1
```

视图还支持集合运算，以弥补字典功能上的不足。

```
>>> a = dict(a = 1, b = 2)
>>> b = dict(c = 3, b = 2)

>>> ka = a.keys()
>>> kb = b.keys()
```

```
>>> ka & kb                    # 交集：在 a、b 中同时存在
{'b'}

>>> ka | kb                    # 并集：在 a 或 b 中存在
{'a', 'b', 'c'}

>>> ka - kb                    # 差集：仅在 a 中存在
{'a'}

>>> ka ^ kb                    # 对称差集：仅在 a 或仅在 b 中出现，相当于“并集-交集”
{'a', 'c'}
```

利用视图集合运算，可简化某些操作。例如，只更新，不新增。

```
>>> a = dict(a = 1, b = 2)
>>> b = dict(b = 20, c = 3)

>>> ks = a.keys() & b.keys()                # 交集，也就是 a 中必须存在的主键

>>> a.update({k: b[k] for k in ks})         # 利用交集结果提取待更新的内容

>>> a
{'a': 1, 'b': 20}
```

扩展

在标准库中，还有几个扩展类型的字典可供使用。

默认字典（defaultdict）类似于 setdefault 包装。当主键不存在时，调用构造参数提供的工厂函数返回默认值。

> 将字典直接作为对外接口时，无法保证用户是否会调用 setdefault 或 get 方法。这样，默认字典的内置初始化行为就好于对用户做额外要求。

```
>>> d = collections.defaultdict(lambda: 100)

>>> d["a"]
100
```

```
>>> d["b"] += 1

>>> d
{'a': 100, 'b': 101}
```

与内部实现无关，有序字典（OrderedDict）明确记录主键首次插入的次序。

> 任何时候都要避免依赖内部实现，或者说遵循“显式优于隐式”的规则。

```
>>> d = collections.OrderedDict()

>>> d["z"] = 1
>>> d["a"] = 2
>>> d["x"] = 3

>>> for k, v in d.items(): print(k, v)
z 1
a 2
x 3
```

与前面所说不同，计数器（Counter）对于不存在的主键返回零，但不会新增。

> 可通过继承并重载__missing__方法新增键值。

```
>>> d = collections.Counter()

>>> d["a"]
0

>>> d["b"] += 1

>>> d
Counter({'b': 1})
```

链式字典（ChainMap）以单一接口访问多个字典内容，其自身并不存储数据。读操作按参数顺序依次查找各字典，但修改操作（新增、更新、删除）仅针对第一字典。

```
>>> a = dict(a = 1, b = 2)
>>> b = dict(b = 20, c = 30)

>>> x = collections.ChainMap(a, b)
```

```
>>> x["b"], x["c"]                              # 按顺序命中
2  30

>>> for k, v in x.items(): print(k, v)          # 遍历所有字典
b 2
c 30
a 1
```

```
>>> x["b"] = 999                                # 更新，命中第一字典
>>> x["z"] = 888                                # 新增，命中第一字典

>>> x
{'a': 1, 'b': 999, 'z': 888}, {'b': 20, 'c': 30}
```

可利用链式字典设计多层次上下文（context）结构。

合理上下文类型，须具备两个基本特征。首先是继承，所有设置可被调用链的后续函数读取。其次是修改仅针对当前和后续逻辑，不应向无关的父级传递。如此，链式字典查找次序本身就是继承体现。而修改操作被限制在当前第一字典中，自然也不会影响父级字典的同名主键设置。

```
>>> root = collections.ChainMap({"a": 1})

>>> child = root.new_child({"b": 200})
>>> child["a"] = 100

>>> child
ChainMap({'b': 200, 'a': 100}, {'a': 1})

>>> child.parents
ChainMap({'a': 1})
```

2.2.7 集合

集合存储非重复对象。所谓非重复，是指除不是同一对象外，值也不能相等。

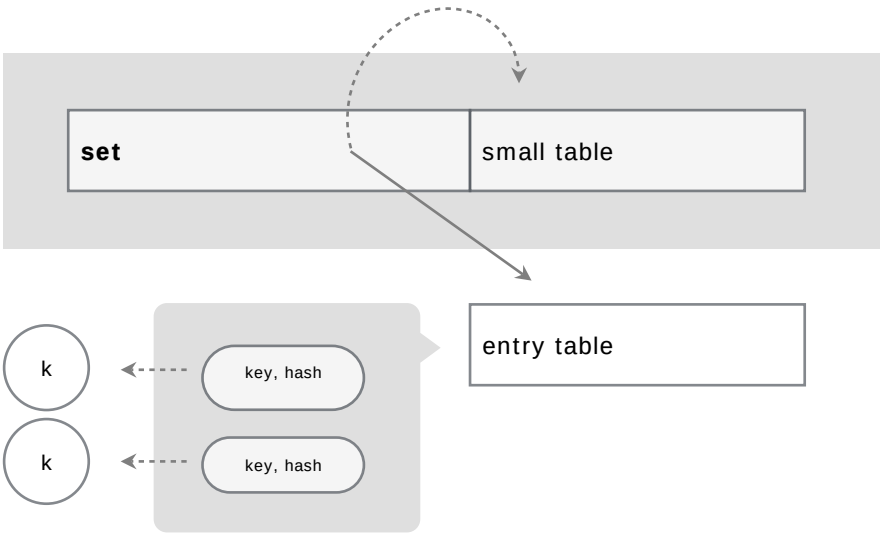
判重公式

```
(a is b) OR (hash(a) == hash(b) AND a == b)
--------    ------------------------------
```

> 如果不是同一对象，那么先判断哈希值，然后比较内容。受限于哈希算法，不同内容可能返回相同的哈希值（哈希碰撞），那么就有必要继续比较内容是否相同。
>
> 为什么先比较哈希值，而不直接比较内容呢？首先，与大多数内容（例如字符串）相比，整数哈希值比较的性能高得多；其次，哈希值不同，内容肯定不同，没有必要继续比较内容。

```
>>> a = 1234
>>> b = 1234

>>> a is b                  # a、b 内容相同，但非同一对象
False

>>> s = {a}                 # 创建集合，初始化元素 a
>>> b in s                  # 使用内容相同的 b 进行判重
True
```

按操作方式，集合分可变（set）和不可变（frozenset）两个版本，内部实现完全相同。用数组实现的哈希表存储元素对象引用，这也就要求元素必须为可哈希类型。



查找元素对象时，先通过算法定位数组索引，继而比较哈希值和内容。

集合对象自带一个长度为 8 的小数组（small table），这对多数简单集合运算有益，可避免额外的内存分配。只有超出容量限制时，才分配更大的数组内存（entry table）。集合使用频率不及列表和字典，内部没采用缓存复用策略。其实现方式决定了无序存储，标准库也没有提供有序实现。

创建

和字典一样使用大括号语法，但初始化数据非键值对。

```
>>> type({})                    # 没有初始化值，表示创建一个空字典
dict

>>> type({"a":1})               # 字典：键值对
dict
```

```
>>> type({1})                   # 集合
set
```

调用类型构造方法创建，或使用推导式。

```
>>> set((1, "a", 1.0))
{1, 'a'}

>>> frozenset(range(3))
frozenset({0, 1, 2})

>>> {x + 1 for x in range(6) if x % 2 == 0}
{1, 3, 5}
```

允许在不同版本间转换。

```
>>> s = {1}
>>> f = frozenset(s)

>>> set(f)
{1}
```

操作

支持大小、相等运算符。

```
>>> {1, 2} > {2, 1}
False

>>> {1, 2} == {2, 1}
True
```

子集判断不能使用 in、not in 语句，它仅用来检查是否包含某个元素。

```
>>> {1, 2} <= {1, 2, 3}           # 子集: issubset
True

>>> {1, 2, 3} >= {1, 2}           # 超集: issuperset
True
```

```
>>> {1, 2} in {1, 2, 3}           # 判断是否包含{1,2}单一元素
False
```

集合作为初等数学概念，其重点自然是并差集运算。合理使用这些操作，可简化算法筛选逻辑，使其具备更好的可读性。

交集：同时属于 A、B 两个集合。

并集：A、B 的所有元素。

差集：仅属于 A，不属于 B。

对称差集：仅属于 A，加上仅属于 B，相当于“并集 - 交集”。

```
>>> {1, 2, 3} & {2, 3, 4}            # 交集: intersection
{2, 3}

>>> {1, 2, 3} | {2, 3, 4}            # 并集: union
{1, 2, 3, 4}

>>> {1, 2, 3} - {2, 3, 4}            # 差集: difference
{1}

>>> {1, 2, 3} ^ {2, 3, 4}            # 对称差集: symmetric_difference
{1, 4}
```

集合运算还可与更新操作一起使用。

```
>>> x = {1, 2}
>>> x |= {2, 3}                 # update

>>> x
{1, 2, 3}
```

```
>>> x = {1, 2}
>>> x &= {2, 3}                 # intersection_update

>>> x
{2}
```

删除操作 remove 可能引发异常，可改用 discard。

```
>>> x = {2, 1}

>>> x.remove(2)
>>> x.remove(2)
```

```
KeyError: 2

>>> x.discard(2)
```

自定义类型

自定义类型虽然是可哈希类型，但默认实现并不足以完成集合去重操作。

```
class User:
    def __init__(self, uid, name):
        self.uid = uid
        self.name = name
```

```
>>> issubclass(User, collections.Hashable)
True
```

```
>>> u1 = User(1, "user1")
>>> u2 = User(1, "user1")

>>> s = set()
>>> s.add(u1)
>>> s.add(u2)

>>> s
{<User at 0x10b69c780>, <User at 0x10b2eb438>}
```

根本原因是默认实现的__hash__方法返回随机值，而__eq__仅比较自身。为符合逻辑需要，须重载这两个方法。

```
class User:

    def __init__(self, uid, name):
        self.uid = uid
        self.name = name

    def __hash__(self):                          # 针对 uid 去重，忽略其他字段
        return hash(self.uid)

    def __eq__(self, other):
        return self.uid == other.uid
```

```
>>> u1 = User(1, "user1")
>>> u2 = User(1, "user2")

>>> s = set()
>>> s.add(u1)
>>> s.add(u2)

>>> s
{<User at 0x10afd8b00>}

>>> u1 in s
True

>>> u2 in s                                    # 仅检查 uid 字段
True
```

第 3 章　表达式

3.1　词法

交互式环境常用来学习和测试，甚至替代系统命令行做些管理工作，但其着实不适合做正式开发。鉴于程序的复杂度和篇幅，很难保证一次性完成编码。事实上，任何优秀的代码，都是在频繁修改和长期调整后才会出现。一个便捷、带有辅助插件，且能提升开发效率的编辑器对于开发人员是很重要的。

个人倾向于用 IPython 作为学习和测试工具，主要是因为其具有非连贯性的操作方式，允许我们实时从不同角度观察目标的各个细节。而用编辑器时，更多的是编写完整逻辑。或者说交互式操作方式属于调试器范畴，与 PDB 工作方式类似。

当然，对于系统性学习而言，Jupyter Notebook 要更方便些，其既有交互式操作，又能编辑说明文字，而且最终以内嵌可执行代码的笔记形式留存下来。

正式可分发的程序，由一到多个源码文件组成，各自对应一个运行期模块。模块内有很多语句，用于定义类型，创建实例，执行逻辑指令。其中包含表达式，可完成数据计算和函数调用。

表达式（expression）由标识符、字面量和操作符组成。其完成运算、属性访问，以及函数调用等。表达式像数学公式那样，总是返回一个结果。

语句（statement）则由一到多行代码组成，其着重于逻辑过程，完成变量赋值、类型定义，以及控制执行流方向等。说起来，表达式算是语句的一种，但语句不一定是表达式。

简单归纳：表达式完成计算，语句执行逻辑。

3.1.1 源文件

如果以 Python 2 为起点，那么下面这个错误便可能是第一只拦路虎，事故源头要追溯到远古时代。在很长时间内，解释器都以 ASCII 为默认编码，如果源码里出现 Unicode 字符，就会导致其无法正确解析。

```
print "您好"
```

输出：

```
$ python2 test.py

SyntaxError: Non-ASCII character ..., but no encoding declared; ...
```

解决方法是在文件头部添加专门的编码声明，指示解释器按此格式读取。

```
# -*- coding: utf-8 -*-

print "您好"
```

好在 Python 3 将默认编码改为 UTF-8，免去了我们为每个源码文件添加此类信息的麻烦。当然，如果你打算使用其他编码格式，则依然需要额外声明。

执行

启动程序，只需将入口文件名作为命令行参数即可。

```
$ python3 main.py
```

或者，在入口文件头部添加 Shebang 信息，指示系统程序载入器用指定解释器执行。但别忘了赋予该文件可执行权限。

main.py

```
#!/usr/bin/env python

print("您好")
```

输出：

```
$ chmod a+x main.py

$ ./main.py
```

考虑到不同系统安装方式上的差异，不建议使用类似“/usr/bin/python”这样的绝对路径，而是以 env 从当前环境设置里查找与之匹配的解释器。当然，还可指定解释器版本，以及相关执行参数。

> 系统命令 env 通过环境变量 PATH 查找目标，这对于使用 VirtualEnv 之类的虚拟环境有益。

```
#!/usr/bin/env python3 -O
```

命令行

命令行参数分解释器和程序两种，分别以 sys.flags、sys.argv 读取。

```
import sys

print(sys.flags.optimize)                    # 解释器参数
print(sys.argv)                              # 程序启动参数
```

输出：

```
$ python -OO main.py 1 2 "hello, world"
```

```
2
['main.py', '1', '2', 'hello, world']
```

对于简单测试代码，无须创建文件或启用交互环境，直接在命令行以-c 参数执行即可。

```
$ python -c "import sys; print(sys.platform)"
darwin

$ python -c "import sys; print(sys.version_info)"
major=3, minor=6, micro=1, releaselevel='final', serial=0
```

退出

终止进程的正式做法是调用 sys.exit 函数，它确保退出前完成相关的清理操作。

> 常见的清理操作包括 finally 和 atexit。前者是结构化异常子句，无论异常是否发生，它总被执行。而 atexit 用于注册在进程退出前才执行的清理函数。

终止进程应返回退出状态码（exit status），以便命令行管理工具据此做出判断。依惯例返回零表示正常结束，其他值为错误。

> 标准库 os 模块里有部分平台的常用定义，但也可自行设定，以表达不同结果。
>
> 辅助函数 exit、quit 由 site 模块创建，适用于交互式环境。不建议在源文件中使用它，这可能导致该文件在 IPython 等环境下执行出错。至于 os._exit，会立即终止进程，不执行任何清理操作。

```
import atexit
import sys

atexit.register(print, "atexit")

try:
    sys.exit()
finally:
```

```
    print("finally")
```

输出：

```
finally
atexit
```

```
$ python -c "import os; exit(os.EX_DATAERR)"; echo $?        # 使用 os 定义
65

$ python -c "exit('error')"; echo $?                         # 非整数退出码
error
1
```

3.1.2 代码

在软件的整个开发和运营周期中，我们面对代码的时间远超过最终部署的可执行系统。若要评述，代码与运营各取其半，乃是极重要的资产。

对开发人员而言，编写代码不仅仅是为了实现逻辑需求，完成日常任务。作为一种另类的记述文档，代码同样要有所追求。如同写文章，不应使用恢宏架构和过度设计来体现优雅，而应追求最自然的逻辑抽象、最简单的文字直述。

综上所述，可阅读性和可测试性是代码的基本要求。前者保证代码自身的可持续性发展，后者维护其最终价值。

缩进

对于强制缩进规则，批评者和欣赏者各持己见。可不管怎么说，强约束规则总好过流于形式的规范手册，这对新手培养严谨的编码风格尤为重要。

PEP8 推荐使用 4 个半角空格表达缩进，但也有人习惯使用制表符。空格可保证在大多数环境下的风格一致，而制表符则可依个人喜好设定显示宽度，算是各有所长。好在，有很多工具能自动转换，算不上什么大问题。

PEP : Python Enhancement Proposals

PEP8 : Style Guide for Python Code

代码格式检查可使用 pycodestyle，或 autopep8、isort、yapf 等增强改进工具。

Python 2 允许在同一文件里混用两种风格，但 Python 3 只能选其一。建议选用能自动识别并转换格式的编辑器，避免出现非语法性错误。

```
def add(a, b):
    z = a + b               # tab 缩进
    return z                # space 缩进，与 tab 保持等宽

add(1, 2)
```

输出：

```
$ python2 -t test.py                                        # 参数-t 检查缩进风格是否一致
inconsistent use of tabs and spaces in indentation

$ python3 test.py
TabError: inconsistent use of tabs and spaces in indentation
```

缩进的另一麻烦在于格式丢失导致逻辑性错误。

```
def sum(x):
    n = 0
    for i in x:
        n += i
        return n                # 缩进错误，导致逻辑错误

print(sum(range(10)))
```

输出：

```
0
```

鉴于 Python 没有代码块结束标志，我们可添加注释作为排版和检查标记。当出现问题

时，该标记能让我们进行手工调整，这对因网页粘贴而导致混乱的大段代码尤其有用。

```
def sum(x):
    n = 0
    for i in x:
        n += i
    # end_for                    # 注释表示块结束。虽对工具无效，但肉眼能发现问题所在
    return n
```

如嫌注释不够优雅，还可创建伪关键字作为结束符号。

```
import builtins
builtins.end = None              # 在内置模块，为 None 添加一个别名。确保后续的任意模块都能使用
```

```
def sum(x):
    n = 0
    for i in x:
        n += i
    end                          # 块结束符号
    return n
```

> 也许是 Python 编译器不够聪明，其不会将这个无意义的 end 忽略掉。不过，与好处相比，它对性能的影响微乎其微。当然，如你有严格的编码习惯，则可用空行区分不同代码块。

语句

无论是为了更好的阅读体验，还是便于在不同环境下编辑，代码行都不宜过长。通常，每条语句独占一行。仅在必要时，以分号分隔多条语句。

但时常也有单语句超过限宽的情况。此时相比于编辑器自动换行，手工硬换行的可读性要更好，因为可调整缩进对齐。即便没有超限，将多条件分成多行，也易于修改，例如调整优先次序，或临时注释掉某些条件。

> 大多数编码规范都将行宽限制在 80 字符。但考虑到该规则源自早期低分辨率显示环境，现在可适当放宽到 100 字符。
>
> 反斜线续行符后不能有空格和注释。

```
if (0 < a < 10)     and \
   (b is not None)  and \
   (c in b):
```

当然，那些有成对括号的表达式不用续行符就可分成多行。

```
a = [
    1,                      # 这里没有续行符，可以有注释
    2,                      # 多行可以方便调整顺序，或注释掉某些初始值
    3,
]

b = {
    "a": 1,
    "b": 2,
}
```

```
def long_function_name(                          # 利用多行将参数分组
        var_one, var_two, var_three,
        var_four):
    pass


long_function_name(var_one, var_two, var_three,
                   var_four)
```

注释

注释作为代码的有益补充而存在，决不能与之相悖。其以自然语言进行描述，力求简洁。注释针对当前代码，言之有物，无多余赘述。应控制注释数量，避免影响代码的修改。更不应画蛇添足，对无歧义的内容多做说明。

在修改代码时，须同步更新注释，以避免造成误解。

注释以井号开头，作用到行尾，分为块注释（block）和内联注释（inline）两类。块注释与代码块平级缩进，用于描述整块代码的逻辑意图和算法设计。内联注释在代码行尾部，补充说明其作用。

> 多数编码规范中要求使用英文注释，但本书为阅读方便，依然主要采用中文注释。

```
def test():

    # block comment
    # line 2
    # line 3
    print()

    x = 1           # inline comment
```

帮助

与被编译器忽略的注释不同，帮助属于基本元数据，可在运行期查询和输出。除在交互环境手工查看外，还用于编辑器智能提示，改善编码体验，或导出生成开发手册。

从表面上来看，帮助就是简单的字符串字面量。考虑到跨行的需要，建议总是用三引号格式。

> 建议为导出成员（public）添加帮助（docstrings）信息。
>
> 文档生成可使用 pydoc，或更专业的 Sphinx。

test.py

```
"""
模块帮助信息
"""


def test():
    """
    函数帮助信息
```

```
    """
    pass
```

> 模块帮助信息不能放到 shebang 前面。

帮助信息保存在__doc__属性里，可直接输出，或以 help 函数查看。

```
>>> import test

>>> test.__doc__
模块帮助信息

>>> test.test.__doc__
函数帮助信息
```

```
>>> help(test)

NAME
    test - 模块帮助信息

FUNCTIONS
    test()
        函数帮助信息
```

当解释器以“OO”优化方式运行时，帮助信息被移除。

```
$ python -c "import test; print(test.__doc__)"
模块帮助信息
```

```
$ python -OO -c "import test; print(test.__doc__)"
None
```

3.2 赋值

前面提及，赋值操作为名字和目标对象建立关联。但这只是其结果，或者说基本用途。在漫长的演化时间里，Python 收集了许多操作便捷但委实让人眼花缭乱的语法风格。

> 对应用开发来说，丰富而便捷的语言功能可提升开发效率。但凡事有度，应限制花式代码和魔法实现。任何时候，代码的可读性都须优先保障。

同时为多个名字建立关联。

```
>>> a = b = c = 1234
>>> a is b is c                  # 引用同一目标
True
```

仅以逗号分隔的多个右值被视作元组的初始化元素。

```
>>> x = 1, "abc", [10, 20]
>>> x
(1, 'abc', [10, 20])
```

3.2.1 增量赋值

顾名思义，增量赋值试图直接修改原对象内容，实现累加效果。当然，其前提是目标对象允许，否则会退化为普通赋值。

既然是增量，就须确保目标对象已存在。

```
>>> s += 10
NameError: name 's' is not defined
```

```
>>> s = 0
>>> s += 10
```

分别以可变列表、不可变元组对比增量赋值的结果。

```
>>> a = b = []
>>> a += [1, 2]

>>> a is b                    # 依然指向原对象
True
```

```
>>> c = d = ()
```

```
>>> c += (1, 2)

>>> c is d                    # 新对象
False
```

同是增量赋值运算，但结果却全然不同。列表直接修改原内容，而元组新建对象。尽管编译器都处理成 INPLACE_ADD 指令，但最终执行还是依目标类型而定。以+=为例，它对应__iadd__方法，这是能否执行增量操作的关键。如该方法不存在，则尝试执行__add__，变成普通加法操作。

```
>>> dis.dis(compile("a += [1, 2]", "", "exec"))
  1           0 LOAD_NAME                0 (a)
              2 LOAD_CONST               0 (1)
              4 LOAD_CONST               1 (2)
              6 BUILD_LIST               2
              8 INPLACE_ADD
             10 STORE_NAME               0 (a)
```

```
>>> dis.dis(compile("a += (1, 2)", "", "exec"))
  1           0 LOAD_NAME                0 (a)
              2 LOAD_CONST               3 ((1, 2))
              4 INPLACE_ADD                                  # 并不能完成增量赋值
              6 STORE_NAME               0 (a)
```

```
>>> "__iadd__" in dir([1, 2])
True

>>> "__iadd__" in dir((1, 2))
False

>>> "__add__" in dir((1, 2))
True
```

3.2.2 序列解包

不同于将多个名字关联到单一对象，序列解包展开所有元素，继而分别与多个名字关联。

```
>>> a, b, c = [1, 2, 3]
>>> a, b, c
1, 2, 3
```

```
>>> a, b, c = "xyz"
>>> a, b, c
'x', 'y', 'z'
```

```
>>> a, b, c = range(3)
>>> a, b, c
0, 1, 2
```

下面的示例可能会造成误解，须注意右值为单一元组对象。

```
>>> a, b = [1, 2], (3, 4)        # 右值表达式构建元组对象([1, 2], (3, 4))

>>> a
[1, 2]

>>> b
(3, 4)
```

解包操作还可用来交换变量，且无须借助第三方。

> 对 3 个以内的变量交换，编译器优化成 ROT 指令，直接交换栈帧数据，而不是构建元组。

```
>>> a, b = b, a
```

```
>>> dis.dis(compile("a, b, c = c, b, a", "", "exec"))
  1           0 LOAD_NAME                0 (c)
              2 LOAD_NAME                1 (b)
              4 LOAD_NAME                2 (a)
              6 ROT_THREE                              # 使用 ROT 指令直接操作栈帧数据
              8 ROT_TWO
             10 STORE_NAME               2 (a)
             12 STORE_NAME               1 (b)
             14 STORE_NAME               0 (c)
```

```
>>> dis.dis(compile("a, b, c, d = d, c, b, a", "", "exec"))
```

```
  1           0 LOAD_NAME                0 (d)
              2 LOAD_NAME                1 (c)
              4 LOAD_NAME                2 (b)
              6 LOAD_NAME                3 (a)
              8 BUILD_TUPLE              4              # 构建元组，解包后分别赋值
             10 UNPACK_SEQUENCE          4
             12 STORE_NAME               3 (a)
             14 STORE_NAME               2 (b)
             16 STORE_NAME               1 (c)
             18 STORE_NAME               0 (d)
```

支持深度嵌套展开。

> 左右值表达式以相同的方式嵌套。

```
>>> a, (b, c) = 1, [10, 20]                    # 简单嵌套方式
>>> a, b, c
1, 10, 20
```

```
>>> a, ((b, c), (d, e)) = 1, [(10, 20), "ab"]       # 更深层次的嵌套
>>> a, b, c, d, e
1, 10, 20, 'a', 'b'
```

当然，也可忽略某些元素。

```
>>> a, _, b, _, c = "a0b0c"
>>> a, b, c
'a', 'b', 'c'
```

星号收集

当序列元素与名字数量不等时，解包出错。

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack (expected 2)

>>> a, b, c = 1, 2
ValueError: not enough values to unpack (expected 3, got 2)
```

如元素多于名字，则可将右值截取等长切片。但反过来就比较麻烦了，需要写代码进行判断。为此，Python 3 专门实现了扩展方式。

> 在名字前添加星号，表示收纳所有剩余元素。

```
>>> a, *b, c = range(5)
>>> a, b, c
0, [1, 2, 3], 4
```

```
>>> a, b, *c = range(5)
>>> a, b, c
0, 1, [2, 3, 4]
```

```
>>> *a, b, c = range(5)
>>> a, b, c
[0, 1, 2], 3, 4
```

即便名字多于元素，也能处理。

```
>>> a, *b, c = 1, 2                # 收集不到数据，返回空列表
>>> a, b, c
1, [], 2
```

注意，解包操作优先保障对非收集名字的赋值，所以元素不能少于非收集名字的数目。另外，星号只能有一个，否则无法界定收集边界。

```
>>> a, *b, c, d = 1, 2
ValueError: not enough values to unpack (expected at least 3, got 2)
```

```
>>> a, *b, *c, d = range(10)
SyntaxError: two starred expressions in assignment
```

星号收集不能单独出现：要么与其他名字一起，要么放入列表或元组内。

```
>>> *a = 1, 2
SyntaxError: starred assignment target must be in a list or tuple
```

```
>>> [*a] = 1, 2
>>> a
[1, 2]
```

```
>>> (*a,) = 1, 2              # 注意是元组，别忘了逗号
>>> a
[1, 2]
```

序列解包和星号收集还用于控制流表达式等场合。

```
>>> for a, *b in ["abc", (1, 2, 3)]: print(a, b)

a ['b', 'c']
1 [2, 3]
```

星号展开

星号还可用来展开可迭代（iterable）对象。

> 简而言之，可迭代对象就是每次返回一个成员。
>
> 所有序列类型，以及字典、集合、文件等都是可迭代类型。

```
>>> a = (1, 2)
>>> b = "ab"
>>> c = range(10, 13)

>>> [*a, *b, *c]
[1, 2, 'a', 'b', 10, 11, 12]
```

对于字典，单星号展开主键，双星号展开键值。

```
>>> d = {"a": 1, "b": 2}

>>> [*d]
['a', 'b']

>>> {"c": 3, **d}
{'c': 3, 'a': 1, 'b': 2}
```

同样可用于函数调用，将单个对象分解成多个实参。

```
def test(a, b, c):
    print(locals())
```

```
>>> test(*range(3))
{'c': 2, 'b': 1, 'a': 0}

>>> test(*[1, 2], 3)
{'c': 3, 'b': 2, 'a': 1}
```

```
>>> a = {"a": 1, "c": 3}
>>> b = {"b": 2}

>>> test(**b, **a)
{'c': 3, 'b': 2, 'a': 1}
```

3.2.3 作用域

作为隐式规则，赋值操作默认总是针对当前名字空间。

> 同一作用域内，名字总属于单一名字空间，不会因执行顺序将其引用到不同名字空间。

```
>>> x = 10

>>> def test():
        print(x)
        x = x + 10

>>> test()
UnboundLocalError: local variable 'x' referenced before assignment
```

```
>>> dis.dis(test)
  2           0 LOAD_GLOBAL              0 (print)
              2 LOAD_FAST                0 (x)            # 本地
              4 CALL_FUNCTION            1
              6 POP_TOP

  3           8 LOAD_FAST                0 (x)            # 本地
```

```
             10 LOAD_CONST               1 (10)
             12 BINARY_ADD
             14 STORE_FAST               0 (x)             # 本地
             16 LOAD_CONST               0 (None)
             18 RETURN_VALUE
```

从反汇编结果看，函数 test 内的 x 统统从本地名字空间引用。并非如设想般，先从全局读取，然后在本地建立关联。

如要对外部变量赋值，须显式声明变量位置。关键字 global 指向全局名字空间，nonlocal 为外层嵌套（enclosing）函数。

除非必要，否则应避免直接对外部变量赋值。可用返回值等方式，交由持有者处理。

```
>>> g = 1

>>> def outer():
        e = 2

        def inner():
            global g                # 声明全局变量
            nonlocal e              # 声明外层嵌套函数变量
            g = 10
            e = 20

        inner()
        return e

>>> outer()
20

>>> g
10
```

显式声明指示编译器生成对外部空间的操作指令。

```
>>> def test():
        global x
```

```
        x = 10

>>> dis.dis(test)
  3           0 LOAD_CONST               1 (10)
              2 STORE_GLOBAL             0 (x)          # 全局
              4 LOAD_CONST               0 (None)
              6 RETURN_VALUE
```

可用 global 在函数内创建全局变量。

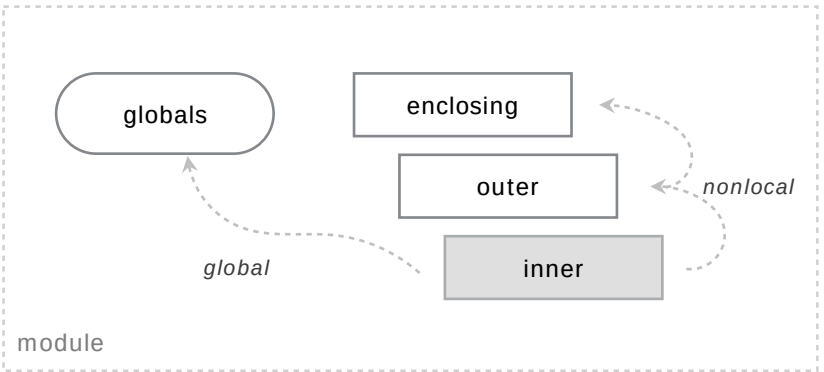
```
>>> x
NameError: name 'x' is not defined          # 检查 x 是否存在

>>> def test():
        global x
        x = 100

>>> test()
>>> x
100
```

至于 nonlocal 则自内向外依次检索嵌套函数，但不包括全局名字空间。

如多层嵌套函数存在同名变量，按就近原则处理。

另，nonlocal 不能为外层嵌套函数新建变量。

```
>>> def enclosing():
        x = 1

        def outer():
            def inner():
                nonlocal x
                x = 999             # 向外查找，直到 enclosing.x 命中

            inner()

        outer()
        print(x)

>>> enclosing()
999
```



不同于 global 运行期行为，nonlocal 要求在编译期绑定，所以目标变量须提前存在。

```
>>> def inner():
        nonlocal x

SyntaxError: no binding for nonlocal 'x' found
```

```
>>> def outer():
        def inner():
            nonlocal x

SyntaxError: no binding for nonlocal 'x' found
```

> 作为写操作的赋值，其规则与读操作 LEGB 完全不同，注意区别对待。

3.3 运算符

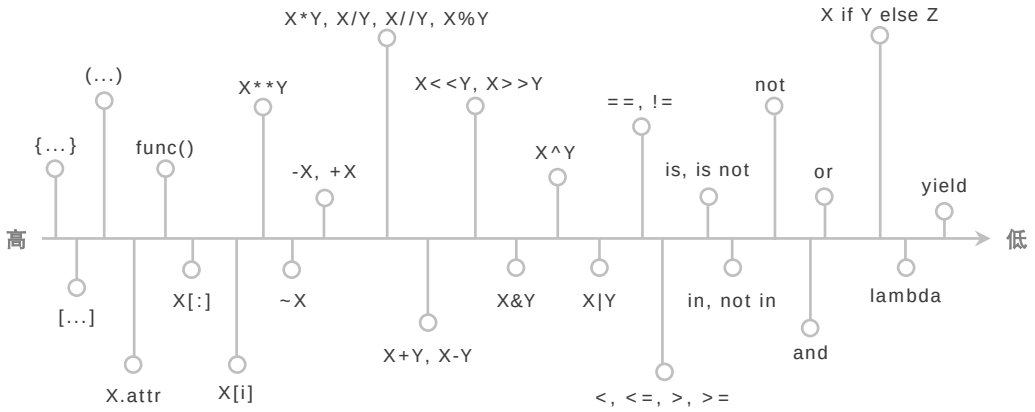
相比于其他语言，Python 运算符更接近自然表达方式。不过，优先级顺序和隐式转换依旧会导致某些隐蔽错误，大家不易察觉。

看看下面的示例，如果不用括号，只怕大家会糊涂得很。

```
>>> not "a" in ["a", 1]                # 谁先谁后?
False
```

```
>>> (not "a") in ["a", 1]              # 相同的返回值可能导致测试盲区
False                                  # 当数据发生变化时，结果是否一致?

>>> not ("a" in ["a", 1])
False
```

适当使用括号，不但可避免不必要的错误，还能提高代码的可读性。



每个运算符都以特定函数或方法实现，可像普通对象那样作为参数传递。

```
def calc(x, y, op):
    return op(x, y)
```

```
>>> import operator

>>> calc(1, 2, operator.add)
3

>>> calc(1, 2, operator.mul)
2
```

> 不仅仅是数学运算符，operator、itemgetter、attrgetter 等还可用于索引和成员访问函数。

标准库还提供了辅助函数，用来简化自定义类型运算符重载实现。

> 使用 functools.total_order 装饰器，可基于__eq__、__lt__，自动补全剩余方法。

```
@functools.total_ordering
class X:
    def __init__(self, n):
        self.n = n

    def __eq__(self, o):
        return self.n == o.n

    def __lt__(self, o):
        return self.n < o.n
```

```
>>> a, b = X(1), X(2)

>>> a <= b
True

>>> a >= b
False
```

Python 3 对运算符做了一些调整。

- 移除“<>”，统一使用“!=”运算符。
- 移除 cmp 函数，须自行重载相关运算符的方法。
- 除法“/”表示 True Division，总是返回浮点数。
- 不再支持反引号 repr 操作，须调用同名函数。
- 不再支持非数字类型混合比较，可自定义相关方法。
- 不再支持字典相等以外的比较操作。

3.3.1 链式比较

链式比较将多个比较表达式组合到一起，更符合人类的阅读习惯。该方式可有效缩短代码，并稍稍提升性能。

```
>>> a, b = 2, 3

>>> a > 0 and b > a and b <= 5
True

>>> 0 < a < b <= 5                    # 链式比较的可读性更好，更易维护
True
```

反汇编查看两者的差异。

```
>>> dis.dis(compile("1 < a and a < 2", "", "eval"))
  1           0 LOAD_CONST               0 (1)
              2 LOAD_NAME                0 (a)          # 载入
              4 COMPARE_OP               0 (<)
              6 JUMP_IF_FALSE_OR_POP    14
              8 LOAD_NAME                0 (a)          # 载入
             10 LOAD_CONST               1 (2)
             12 COMPARE_OP               0 (<)
        >>   14 RETURN_VALUE
```

```
>>> dis.dis(compile("1 < a < 2", "", "eval"))
  1           0 LOAD_CONST               0 (1)
              2 LOAD_NAME                0 (a)          # 载入
              4 DUP_TOP                                 # 直接复制
              6 ROT_THREE
              8 COMPARE_OP               0 (<)
             10 JUMP_IF_FALSE_OR_POP    18
             12 LOAD_CONST               1 (2)
             14 COMPARE_OP               0 (<)
             16 RETURN_VALUE
        >>   18 ROT_TWO
             20 POP_TOP
             22 RETURN_VALUE
```

显然，链式比较减少了载入指令，其更多基于栈数据复制和交换。仅凭这一点，其执行

性能就有所提高。但从整体上来看，这点改善远不如代码的可读性和可维护性吸引人。

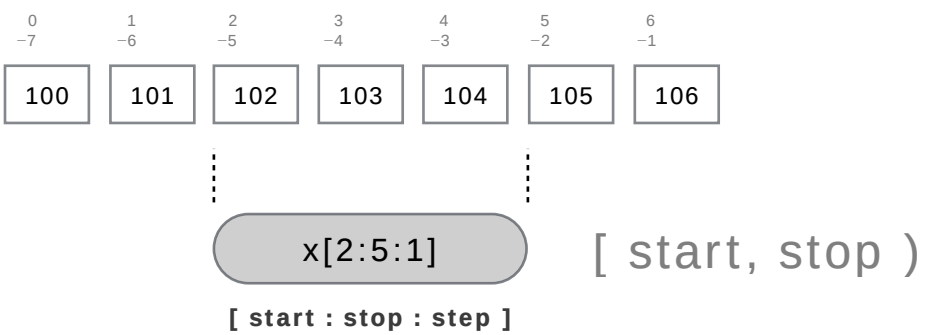
3.3.2 切片

切片用于表达序列片段或整体。具体行为与其在语句中的位置有关，作为右值时复制序列数据，左值则表达操作范围。

```
>>> x = [0, 1, 2, 3, 4, 5, 6]
>>> s = x[2:5]                          # 从列表中复制指定范围的引用

>>> s
[2, 3, 4]
```

```
>>> x.insert(3, 100)                    # 对原列表的修改，不影响切片

>>> x
[0, 1, 2, 100, 3, 4, 5, 6]

>>> s
[2, 3, 4]
```

因列表存储的是元素指针，那么复制的自然也是指针，而非元素对象。切片所返回的新列表与原列表除共享部分元素对象外，其他毫无干系。



完整切片操作由 3 个参数构成。

其以起始和结束索引构成一个半开半闭区间，不含结束位置。

默认的起始位置为 0；结束位置为 len(x)，以容纳最后一个元素。

```
>>> x = [100, 101, 102, 103, 104, 105, 106]

>>> x[2:5:1]
[102, 103, 104]
```

```
>>> x[:5]                                   # 省略起始索引
[100, 101, 102, 103, 104]

>>> x[2:]                                   # 省略结束索引
[102, 103, 104, 105, 106]

>>> x[:]                                    # 完整复制
[100, 101, 102, 103, 104, 105, 106]
```

可指定步进幅度，间隔选取元素。甚至可以是负值，从右至左反向行进。

索引 0 表示正向第一元素，所以反向索引从–1 起始。

```
>>> x[2:6:2]
[102, 104]

>>> x[::2]
[100, 102, 104, 106]
```

```
>>> x[::-1]                                 # 反向步进，全部复制
[106, 105, 104, 103, 102, 101, 100]

>>> x[5:2:-1]                               # 反向步进，使用正索引表示起始、结束位置
[105, 104, 103]

>>> x[-2:-5:-1]                             # 使用负索引表达起始、结束位置
[105, 104, 103]
```

除表达式外，也可用 itertools.islice 函数执行切片操作。

事实上，负索引不仅用于切片，也可直接访问序列元素。

```
>>> (0, 1, 2)[-2]
1
```

删除

用切片指定要删除的序列范围。

```
>>> x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> x[3:7]                                  # 查看要删除的元素
[3, 4, 5, 6]

>>> del x[3:7]

>>> x
[0, 1, 2, 7, 8, 9]
```

```
>>> x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> x[3:7:2]                                # 查看要删除的元素
[3, 5]

>>> del x[3:7:2]                            # 步进删除

>>> x
[0, 1, 2, 4, 6, 7, 8, 9]
```

赋值

以切片方式进行序列局部赋值，相当于先删除，后插入。

```
>>> x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> x[3:7] = [100, 200]
```

```
>>> x
[0, 1, 2, 100, 200, 7, 8, 9]
```

如设定步进，则删除和插入的元素数量必须相等。

```
>>> x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> x[::2]                                          # 查看要删除的元素
[0, 2, 4, 6, 8]

>>> x[::2] = [100, 200, 400, 600, 800]              # 步进插入

>>> x
[100, 1, 200, 3, 400, 5, 600, 7, 800, 9]
```

```
>>> x[::2] = [0, 2, 4]
ValueError: attempt to assign sequence of size 3 to extended slice of size 5

>>> x[::2] = [0, 2, 4, 6, 8, 10]
ValueError: attempt to assign sequence of size 6 to extended slice of size 5
```

3.3.3 逻辑运算

逻辑运算用于判断多条件的布尔结果，或返回有效的操作数。

分别以 and、or、not 运算符表示逻辑与、或、非三种关系。

其中 and 返回最后，或导致短路的操作数；or 返回第一真值，或最后的操作数。

```
>>> 1 and 2                     # 最后的操作数
2

>>> 1 and 0 and 2               # 导致短路的操作数
0
```

```
>>> 1 or 0                      # 第一真值
1

>>> 0 or 1 or 2                 # 第一真值
```

```
1

>>> 0 or []                    # 最后的操作数
[]
```

相同逻辑运算符一旦短路，后续计算被终止。

```
def x(o):                      # 输出执行步骤
    print("op:", o)
    return o

>>> x(0) and x(1)              # 0 导致短路
op: 0
0

>>> x(1) or x(2)               # 返回 1 真值，短路
op: 1
1
```

反汇编可以看得更清楚些。

```
>>> dis.dis(compile("0 and 1 and 2 and 3", "", "eval"))
  1           0 LOAD_CONST               0 (0)
              2 JUMP_IF_FALSE_OR_POP    14                 # 如果为 False，就跳转到 14
              4 LOAD_CONST               1 (1)
              6 JUMP_IF_FALSE_OR_POP    14
              8 LOAD_CONST               2 (2)
             10 JUMP_IF_FALSE_OR_POP    14
             12 LOAD_CONST               3 (3)
        >>   14 RETURN_VALUE
```

```
>>> dis.dis(compile("1 or 2 or 3 or 4", "", "eval"))
  1           0 LOAD_CONST               0 (1)
              2 JUMP_IF_TRUE_OR_POP     14
              4 LOAD_CONST               1 (2)
              6 JUMP_IF_TRUE_OR_POP     14
              8 LOAD_CONST               2 (3)
             10 JUMP_IF_TRUE_OR_POP     14
             12 LOAD_CONST               3 (4)
        >>   14 RETURN_VALUE
```

当然，不同运算符须多次计算。

```
>>> x(0) and x(1) or x(2)
op: 0
op: 2
2
```

```
>>> dis.dis(compile("0 and 1 and 2 or 9", "", "eval"))
  1           0 LOAD_CONST               0 (0)
              2 POP_JUMP_IF_FALSE       12                 # 如果为 False，就跳转到 12
              4 LOAD_CONST               1 (1)
              6 POP_JUMP_IF_FALSE       12
              8 LOAD_CONST               2 (2)
             10 JUMP_IF_TRUE_OR_POP     14
        >>   12 LOAD_CONST               3 (9)
        >>   14 RETURN_VALUE
```

条件表达式

常见逻辑运算是条件表达式，类似功能在其他语言被称作三元运算符。

> T if X else F ：当条件 X 为真时，返回 T，否则返回 F。其等同于 X ? T : F。

```
>>> "T" if 2 > 1 else "F"        # 等同 2 > 1 ? T : F
'T'

>>> "T" if 2 < 1 else "F"
'F'
```

也可用逻辑运算符实现同等效果，且方式更接近传统习惯。

```
>>> 2 > 1 and "T" or "F"          # 2 > 1 ? T : F
'T'

>>> 2 < 1 and "T" or "F"
'F'
```

分解执行步骤，这很容易理解。

```
>>> 2 < 1 and "T" or "F"
'F'

>>> 2 < 1 and "T"                # ① 2<1 导致短路
False

>>> False or "F"                 # ② 返回真值 F
'F'
```

不过此方式存在缺陷：当 T 为假时，or 必然返回最后的操作数，这与预期不符。

```
>>> 2 > 1 and "" or "F"          # T == ""
'F'

>>> 2 > 1 and ""                 # ① 返回最后的操作数""
''

>>> "" or "F"                    # ② 返回 F，与期望值""不符
'F'
```

显然，当 T 和 F 是动态数据时，条件表达式更安全一些。

```
>>> "" if 2 > 1 else "F"
''
```

逻辑运算符还常被用来简化默认值设置。

```
>>> x = None

>>> y = x or 100

>>> y
100
```

```
>>> x = None

>>> y = x and x * 2 or 100

>>> y
100
```

3.4 控制流

用对象构建好舞台和演员，剩下的就是用故事线串联起整个世界。每个场景都由一系列矛盾冲突推动前行，并不总是平淡悠闲。其中有选择，有轮回，还须具有大魄力抽身事外。

3.4.1 选择

语法除 elif 缩写外，其他并无不同。多选择分支依次执行条件表达式，最终全部失败，或仅一条得以执行。

```
def test(x):
    if x > 0:
        print("+")
    elif x < 0:
        print("-")
    else:
        print("=")
```

```
>>> dis.dis(test)
  2           0 LOAD_FAST                0 (x)
              2 LOAD_CONST               1 (0)
              4 COMPARE_OP               4 (>)
              6 POP_JUMP_IF_FALSE       18              # 当前条件失败，转入下一分支

  3           8 LOAD_GLOBAL              0 (print)
             10 LOAD_CONST               2 ('+')
             12 CALL_FUNCTION            1
             14 POP_TOP
             16 JUMP_FORWARD            26 (to 44)      # 分支执行后，立即跳出整个 if 语句块

  4     >>   18 LOAD_FAST                0 (x)
             20 LOAD_CONST               1 (0)
             22 COMPARE_OP               0 (<)

               ...

        >>   44 LOAD_CONST               0 (None)
             46 RETURN_VALUE
```

> 无论单个 if 有多少选择分支，最多仅有一条得以执行。而多个 if 语句，则可能有多条，甚至全部被执行。两者的意义和执行方式完全不同，注意区别。

某些时候，可将 else 分支从选择语句移除，作为默认返回。

```
def test(x):
    if x > 0:
        return 1
    elif x < 0:
        return -1

    return 0
```

应使用各种方式减少选择语句及分支，减少缩进层次，避免流程控制里包含太多的细节。

- 将过长的分支代码重构为函数。相比于细节，有意义的函数名更友好。
- 将复杂或过长的条件表达式重构为函数，更易阅读和维护。
- 代码块跨度太长（比如需要翻屏），极易造成缩进错误。
- 嵌套容易引发缩进混乱，且层次太多可读性也较差，故应避免使用。
- 简单选择语句，用条件表达式或逻辑运算替代。

> 在描述倒车流程时，会说：“刹车，挂挡倒 ……”，并不会加入发动机等部件的工作细节。因流程和细节所处的层次不同，混到一起，未免有主次不分之嫌。
>
> 在阅读和研究代码时，我们更期望有简单清晰的逻辑主线，以便快速了解整个过程，然后才是选择流程中出现的某个函数去探究细节。在逻辑流程中，用友好的名字遮蔽细节是极重要的。
>
> 在下面的示例中，我们尝试让框架的可读性更好，并尽可能消除过多的选择语句和缩进。

伪码

```
def back_car():
    if not check(car_status):                          # 检查条件独立成函数，此处无须了解细节
        raise Exception

    actions = (brake, reversing_light, reverse_gear)   # 用方法表消除过多的 if 语句
```

```
    if not all(map(lambda do: do(), actions)):          # 用 map 代替 for 循环
        raise Exception
```

对新手而言，过多的选择分支易造成死代码。

> 编译器可能对此漠不关心，不做任何提示和消除（DCE）处理，而我们只能借助代码分析检查工具。

```
def test(x):
    if x > 0:
        print("a")
    elif x > 5:                 # 死代码，永远不会被执行
        print("b")
```

```
>>> dis.dis(test)
  2           0 LOAD_FAST                0 (x)
              2 LOAD_CONST               1 (0)
              4 COMPARE_OP               4 (>)
              6 POP_JUMP_IF_FALSE       18

                ...

  4     >>   18 LOAD_FAST                0 (x)            # 死代码的分支代码依然存在
             20 LOAD_CONST               3 (5)
             22 COMPARE_OP               4 (>)
             24 POP_JUMP_IF_FALSE       34

  5          26 LOAD_GLOBAL              0 (print)
             28 LOAD_CONST               4 ('b')
             30 CALL_FUNCTION            1
             32 POP_TOP
        >>   34 LOAD_CONST               0 (None)
             36 RETURN_VALUE
```

3.4.2 循环

循环语句分为 while、for 两种，不存在替代关系。前者用于执行逻辑循环；而后者则偏向于对象内容迭代，其或许改名为 foreach 更易区分和理解。

```
>>> while n > 0:                    # 基于特定条件，循环执行指令
        print(n)
        n -= 1
```

```
>>> for i in range(3):              # 依次从可迭代对象中提取元素
        print(i)
```

迭代

遍历可迭代对象（iterate）时，可进一步执行序列解包。次序是先取迭代元素，随后对该元素做解包操作。

```
>>> for a, b in [(1, 2), (3, 4)]:
        print(a, b)

1 2
3 4
```

类似于下面的错误，纯属误解。

```
>>> for a, b in [1, 2]:                    # 每次从列表取一个元素，单个整数无法展开
        print(a, b)

TypeError: 'int' object is not iterable
```

如要实现传统 for 循环，可借助 enumerate 类型。它为迭代元素添加自增序号，如此解包操作就可获取索引值。

> 无须修改原对象，enumerate 维持一个索引计数器即可。可设定起始序号。

```
>>> for i, x in enumerate([100, 200, 300]):
        print(i, x)

0 100
1 200
2 300
```

如迭代对象以函数返回，那么该函数仅调用一次。

```
>>> def data():
        print("data")
        return range(3)

>>> for i in data():
        print(i)

data                            # 仅执行一次
0
1
2
```

```
>>> dis.dis(compile("for i in data(): pass", "", "exec"))
  1           0 SETUP_LOOP              14 (to 16)
              2 LOAD_NAME                0 (data)
              4 CALL_FUNCTION            0              # 调用 data 函数，将结果存储在堆栈中
              6 GET_ITER
        >>    8 FOR_ITER                 4 (to 14)      # 开始迭代
             10 STORE_NAME               1 (i)
             12 JUMP_ABSOLUTE            8              # 跳转到 8 继续迭代，而非调用 data 函数
        >>   14 POP_BLOCK
        >>   16 LOAD_CONST               0 (None)
             18 RETURN_VALUE
```

可选分支

与其他语言不同，Python 循环语句可自选 else 分支，在循环正常结束时执行。

> 正常结束是指循环没有被 break、return 中断。当然，循环体没被执行也属正常。
>
> 另外，执行 continue 是允许的，它不是中断。

```
>>> n = 3
>>> while n > 0:
        n -= 1
    else:
        print("over.")

over.
```

```
>>> n = 3
>>> while n > 0:
        print("break.")
        break                           # break 导致 else 不会被执行
    else:
        print("over.")

break.
```

```
>>> n = 0
>>> while n > 0:
        break                           # 尽管有 break，但并未被执行，也算正常结束
    else:
        print("over.")

over.
```

该特性可用来处理一些默认行为，比如循环匹配失败时记录日志或引发异常。

我不太喜欢它。首先，这让原本职责单一的循环语句存在两种毫无相干的逻辑；其次，同时用多种语言编程时，存在干扰，并不自然。

和选择语句不同，选择语句肯定只有一条分支被执行，而循环分支可能执行，也可能不执行。这已不仅是语法差异，而是迥然不同的执行方式。除 Pythonic 外，实在想不出有多少实用场景。

```
def match(data, prefix):
    while data:
        if data[0].startswith(prefix): return True     # 找到后，以 return 中断循环
        data = data[1:]
    else:                                              # 没有中断，自然是没找到
        print("log: not found")

    return False
```

```
>>> match(range(3), 4)
log: not found
False
```

临时变量

循环语句没有单独的名字空间，其内部临时变量直接影响所在的上下文。

```
def test():
    while True:
        x = 100
        break

    for i in range(10, 20):
        pass

    print(locals())                 # x、i 实际使用 test.locals 名字空间
    print(x, i)
```

```
>>> test()
{'i': 19, 'x': 100}
100 19
```

跳转

Python 不支持 goto 和 label，想要在多层嵌套循环中跳出，稍有些麻烦。最简单的做法是设定结束标志，然后在相应的位置检查。

```
def test():
    stop = False

    while True:
        for i in range(10):
            stop = True             # 中断前设定跳出标志，供外循环判断
            break

        if stop: break
```

或将内层循环重构为函数，以消除嵌套。

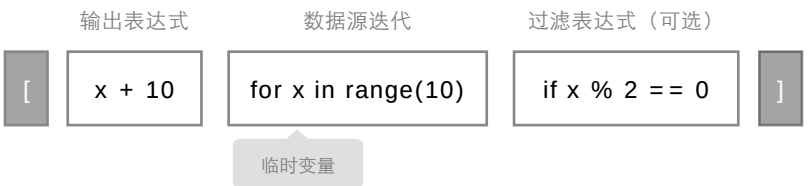
```
def test():
    def inner():                    # 内循环重构函数，以返回值作为跳出标志
        for i in range(10):
            return False
```

```
        return True

    while True:
        if not inner(): break
```

还可抛出异常，在跳出位置捕获即可。具体内容详见后续章节。

3.5 推导式

推导式文法像出自某位数学家之手，其用自然而简便的方式，整合 for、if 语句，用于构造列表、字典和集合对象。




```
>>> [x for x in range(5)]
[0, 1, 2, 3, 4]

>>> [x + 10 for x in range(10) if x % 2 == 0]
[10, 12, 14, 16, 18]
```

将推导式拆解成普通语句，更易理解其执行过程。但对比之下，这显得不够优雅。

```
l = []

for x in range(10):
    if x % 2 == 0: l.append(x + 10)
```

除列表外，还可用大括号创建字典和集合，区别在于输出表达式是否为键值对。

```
>>> {k:v for k, v in zip("abc", range(10, 13))}      # 字典
{'a': 10, 'b': 11, 'c': 12}
```

```
>>> {k:0 for k in "abc"}
{'a': 0, 'b': 0, 'c': 0}
```

```
>>> {x for x in "abc"}                                  # 集合
{'b', 'c', 'a'}
```

还可直接作为函数调用实参。

```
def test(data):
    print(type(data), data)
```

```
>>> test({x for x in range(3)})
<class 'set'> {0, 1, 2}
```

嵌套

改造下面这样一个层次过多的嵌套循环。

```
>>> l = []

>>> for x in "abc":
        if x != "c":
            for y in range(3):
                if y != 0:
                    l.append(f"{x}{y}")

>>> l
['a1', 'a2', 'b1', 'b2']
```

推导式允许有多个 for 子句，每个子句都可选一个 if 条件表达式。

```
>>> [f"{x}{y}" for x in "abc" if x != "c"
               for y in range(3) if y != 0]

['a1', 'a2', 'b1', 'b2']
```

> 两相对比后，推导式绝对是你日后积极使用，且念念不忘的语法。

性能

除了语法因素外，推导式还有性能上的优势。

```
def test_comp(data):
    return [x for x in data]


def test_append(data):
    l = []
    append = l.append                          # 避免查找属性开销

    for x in data: append(x)
    return l
```

使用静态数据源，避免额外干扰。但从测试结果看，差距已然不小，且随数据量递增。

> 比较对象是 for + append，而非以 C 实现的 list 构造。毕竟构造方法不支持过滤表达式。

```
>>> data = list(range(10000))                  # 准备测试数据

>>> %timeit test_comp(data)
240 µs ± 3 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

>>> %timeit test_append(data)
472 µs ± 17.6 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

> 编译器将推导式编译成函数，随后直接以 LIST_APPEND 指令向列表添加数据。如此可减少方法调用的额外开销。

```
>>> dis.dis(test_comp)
  2           0 LOAD_CONST               1 (<code object <listcomp>>)
              2 LOAD_CONST               2 ('test_comp.<locals>.<listcomp>')
              4 MAKE_FUNCTION            0
              6 LOAD_FAST                0 (data)
              8 GET_ITER
             10 CALL_FUNCTION            1
             12 RETURN_VALUE


>>> dis.dis(test_comp.__code__.co_consts[1])
  2           0 BUILD_LIST               0
              2 LOAD_FAST                0 (.0)
        >>    4 FOR_ITER                 8 (to 14)
              6 STORE_FAST               1 (x)
              8 LOAD_FAST                1 (x)
             10 LIST_APPEND              2                 # 指令，而非方法调用
             12 JUMP_ABSOLUTE            4
        >>   14 RETURN_VALUE
```

临时变量

和普通循环语句不同，推导式临时变量不影响上下文名字空间。

```
def test():
    a = "abc"
    data = {a:b for a, b in zip("xyz", range(10, 13))}      # a,b 不会影响 test.locals
    print(locals())
```

```
>>> test()
{'data': {'x': 10, 'y': 11, 'z': 12}, 'a': 'abc'}
```

从输出结果看，推导式临时变量既没有修改外部同名变量，也没有新建关联。

因为推导式变量在编译器自动生成的函数内使用，而非 test 函数。

生成器

在推导式语法中使用小括号，结果并非创建元组，而是创建生成器（generator）对象。区别于推导式，我们称其为生成器表达式。

```
>>> (x for x in range(3))
<generator object <genexpr> at 0x10812f468>
```

第 4 章　函数

4.1　定义

函数是代码复用和模块化的基本单位，用于执行单一逻辑。

相比于语句，函数以整体的形式面向调用者，函数名称、参数列表，还有返回值构成接口原型，其天生带有设计的味道。函数还是代码重构的源头，涉及重命名、参数整理等诸多技巧。莫小看函数，我们应以此为起点，追求精致整洁、优雅简约的风格，写出没有坏味道的代码。

> 整洁的代码简单直接，如同优美的散文。它从不隐藏设计者的意图，充满干净利落的抽象和直截了当的控制语句。
>
> —— Grady Booch

函数应减少依赖关系，具备良好的可测试性和可维护性，这是性能优化的关键所在。另外，我们还应遵循一个基本原则，就是专注于做一件事，不受外在干扰和污染。

> 比如，当某个函数同时具备修改和查询功能时，正确的做法是拆分成查询和修改两个不同函数。从逻辑上讲，查询可能返回零，但修改必然是预设存在条件的。再则，查询性能优化面向缓存，这于修改不利，两者无法共处。

函数要短而精，使用最小作用域。如有可能，应确保其行为的一致性。如果逻辑受参数影响而有所不同，那应该将多个逻辑分支分别重构成独立函数，使其从“变”转为“不变”。如此，函数更易测试，更易扩展。

最后，要及时清理不再使用的参数、代码，以及注释。任何用不到的元素，无论出发点如何，最终也只是拖累。与其覆满霜尘，莫若茅檐长扫，花木成畦。

创建

函数由两部分组成：代码对象持有字节码和指令元数据，负责执行；函数对象则为上下文提供调用实例，并管理所需的状态数据。

> 假定函数是生产线，那么代码对象就是已设定完毕的数控机床。即便如此，机床本身也只是生产工具，不能对外提供服务。需要工厂为其提供场地、电力等环境，并为其承接生产任务，这就是函数对象的角色。当然，同一种机床可卖给不同工厂。

```
def test(x, y = 10):
    x += 100
    print(x, y)
```

```
>>> test                                # 函数对象
<function __main__.test>

>>> test.__code__                       # 代码对象
<code object>
```

代码对象的相关属性由编译器生成，为只读模式。存储指令运行所需的相关信息，诸如源码行、指令操作数，以及参数和变量名等。

```
>>> test.__code__.co_varnames           # 参数及变量名列表
```

```
('x', 'y')

>>> test.__code__.co_consts                # 指令常量
(None, 100)
```

```
>>> dis.dis(test.__code__)
  2           0 LOAD_FAST                0 (x)
              2 LOAD_CONST               1 (100)
              4 INPLACE_ADD
              6 STORE_FAST               0 (x)

  3           8 LOAD_GLOBAL              0 (print)
             10 LOAD_FAST                0 (x)
             12 LOAD_FAST                1 (y)
             14 CALL_FUNCTION            2
             16 POP_TOP
             18 LOAD_CONST               0 (None)
             20 RETURN_VALUE
```

如果是 dis(test.__code__.co_code)，则会看到没有元数据符号的反汇编结果。

与代码对象只关注执行不同，函数对象作为外部实例存在，负责管理运行期状态。比如上例中的参数默认值，以及动态添加的属性等。

```
>>> test.__defaults__                  # 参数默认值
(10,)

>>> test(1)
101 10
```

```
>>> test.__defaults__ = (1234,)        # 修改默认值

>>> test(1)
101 1234
```

```
>>> test.abc = "hello, world"          # 为函数实例添加属性

>>> test.__dict__
{'abc': 'hello, world'}
```

事实上，def 是运行期指令。以代码对象为参数，创建函数实例，并在当前上下文中与指定的名字相关联。

```
>>> dis.dis(compile("def test(): pass", "", "exec"))
  1           0 LOAD_CONST               0 (<code object>)
              2 LOAD_CONST               1 ('test')
              4 MAKE_FUNCTION            0
              6 STORE_NAME               0 (test)
```

伪码

```
test = make_function("test", code)
```

> 反汇编操作，会在函数实例创建后执行。目标针对__code__，而非针对创建过程。

正因如此，可用 def 以单个代码对象为模板创建多个函数实例。

```
def make(n):
    ret = []

    for i in range(n):
        def test(): print("hello")               # test = make_function(code)

        print(id(test), id(test.__code__))
        ret.append(test)

    return ret
```

```
>>> make(3)
4428346232  4425999248                           # 不同实例，相同代码
4430389456  4425999248
4430390136  4425999248

[<function __main__.make.<locals>.test>,
 <function __main__.make.<locals>.test>,
 <function __main__.make.<locals>.test>]
```

多个实例，和多个名字引用同一实例，不是一回事。

用列表持有多实例，阻止临时变量 test 被回收，避免因内存复用而出现相同的 id 值。

在名字空间里，名字仅能与单个目标相关联。如此，就无法实现函数重载（overload）。另外，作为第一类对象（first-class），函数可作为参数和返回值传递。

```
>>> def test(op, x, y):
        return op(x, y)

>>> def add(x, y):
        return x + y

>>> test(add, 1, 2)                    # 将函数作为参数
3
```

```
>>> def test():
        def hello():
            print("hello, world!")

        return hello                    # 将函数作为返回值

>>> test()()
hello, world!
```

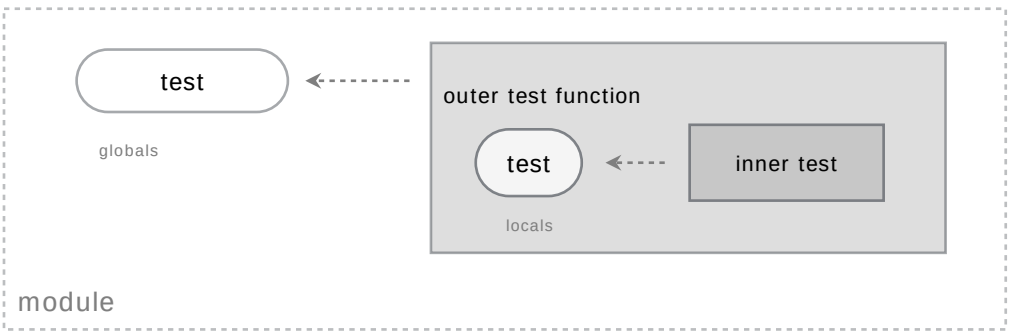
嵌套

支持函数嵌套，其甚至可与外层函数同名。

```
def test():
    print("outer test")

    def test():
        print("inner test")

    return test
```

```
>>> x = test()
outer test

>>> x
<function __main__.test.<locals>.test>

>>> x()
inner test
```

内外函数名字虽相同，但分属不同层次的名字空间。




```
outer
>>> dis.dis(test)
  2           0 LOAD_GLOBAL              0 (print)
              2 LOAD_CONST               1 ('outer test')
              4 CALL_FUNCTION            1
              6 POP_TOP

  3           8 LOAD_CONST               2 (<code object test>)
             10 LOAD_CONST               3 ('test.<locals>.test')
             12 MAKE_FUNCTION            0
             14 STORE_FAST               0 (test)              # 保存到当前名字空间

  5          16 LOAD_FAST                0 (test)              # 从当前名字空间返回
             18 RETURN_VALUE
```

匿名函数

在 Python 里，匿名函数的正式名称为 lambda 表达式。

相比于普通函数，匿名函数的内容只能是单个表达式，而不能使用语句，也不能提供默认函数名，否则其也不会被称为匿名函数了。除此之外，匿名函数的使用方法与普通函数并无差异。

```
>>> add = lambda x, y: x + y
>>> add
<function __main__.<lambda>>

>>> add(1, 2)
3
```

普通函数总有一个默认名字（__name__），用以标识真实身份。该名字是编译期静态绑定的，与运行期的变量名引用无关。

> PEP3155：可用__qualname__获取目标在模块中的完整定义路径。

```
>>> def test(): pass

>>> a = test

>>> a.__name__
'test'

>>> a
<function test at 0x1099b29d8>
```

而 lambda 只有变量引用，没有自己的名字。

```
>>> test = lambda: None

>>> a = test

>>> a.__name__
'<lambda>'
```

就连创建过程都是“路人甲”待遇。

```
>>> dis.dis(compile("def test(): pass", "", "exec"))
```

```
  1           0 LOAD_CONST               0 (<code object test>)       # 有名有姓
              2 LOAD_CONST               1 ('test')                   # 静态名字
              4 MAKE_FUNCTION            0
              6 STORE_NAME               0 (test)                     # 在上下文中使用自己的名字
              8 LOAD_CONST               2 (None)
             10 RETURN_VALUE
```

```
>>> dis.dis(compile("lambda: None", "", "exec"))
  1           0 LOAD_CONST               0 (<code object <lambda>>)   # 没有名字，统统为“路人甲”
              2 LOAD_CONST               1 ('<lambda>')               #  “路人甲”
              4 MAKE_FUNCTION            0                            # 创建完了，直接返回
              6 POP_TOP                                               # 如没有变量赋值，那就走丢了
              8 LOAD_CONST               2 (None)
             10 RETURN_VALUE
```

但在适用场合，lambda 远比普通函数更加灵活和自由。

```
>>> map(lambda x: x ** 2, range(3))                # 直接作为参数
```

```
>>> ops = {
        "add": lambda x, y: x + y,                 # 构建方法表
        "sub": lambda x, y: x - y,
   }

>>> ops["add"](2, 3)
5
```

```
>>> def make(n):
        return [lambda: print("hello") for i in range(n)]    # 作为推导式输出结果
```

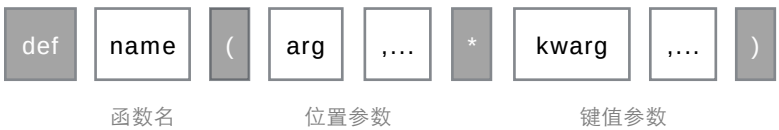
lambda 同样支持嵌套，还可直接调用。

```
>>> test = lambda x: (lambda y: x + y)       # 将另一个 lambda 作为返回值，支持闭包

>>> add = test(2)
>>> add(3)
5
```

```
>>> (lambda x: print(x))("hello")            # 使用括号避免语法错误
hello
```

4.2 参数

按定义和传参方式，参数可分作位置和键值两类。可设置默认值，或定义变参收集多余的参数，但不支持参数嵌套。



> 参数又有形参和实参之分。形参出现在函数定义的参数列表中，可视作函数局部变量，仅能在函数内部使用。而实参由调用方提供，通常以复制方式将值传递给形参。形参在函数调用结束后销毁，而实参则受调用方作用域的影响。不同于形参以变量形式存在，实参可以是变量、常量、表达式等，总之有确定值可供复制传递。
>
> 不管实参是名字、引用，还是指针，其都以值复制方式传递，随后的形参变化不会影响实参。当然，对该指针或引用目标的修改，与此无关。

形参如普通局部变量那样出现在函数的名字空间内。实参按顺序传递，也可以星号展开。

```
def test(a, b, c = 3):
    print(locals())
```

```
>>> test(1, 2)                          # 忽略有默认值的参数
{'c': 3, 'b': 2, 'a': 1}

>>> test(1, 2, 30)                      # 为默认值参数显式提供实参
{'c': 30, 'b': 2, 'a': 1}

>>> test(*(1, 2, 30))                   # 星号展开
{'c': 30, 'b': 2, 'a': 1}
```

以命名方式传递时，无须理会参数顺序。这对于字典展开非常方便。

```
>>> test(b = 2, a = 1)
{'c': 3, 'b': 2, 'a': 1}
```

```
>>> test(**{"b": 2, "a":1})          # 键值展开后，等同于命名传递
{'c': 3, 'b': 2, 'a': 1}
```

混用这两种方式，必须确保顺序在命名参数之前。

```
>>> test(1, c = 30, b = 2)
{'c': 30, 'b': 2, 'a': 1}

>>> test(c = 30, 1, 2)
SyntaxError: positional argument follows keyword argument
```

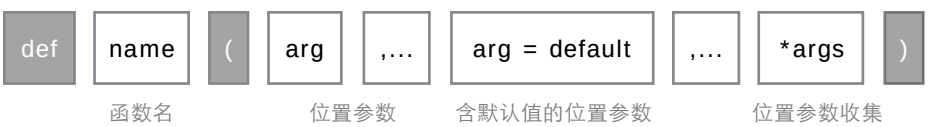
位置参数

位置参数按排列顺序，又可细分为：

1．普通位置参数，零到多个。

2．有默认值的位置参数，零到多个。

3．单星号收集参数，仅一个。



收集参数将多余的参数值收纳到一个元组对象里。所谓多余，是指对普通参数和有默认值的参数全部赋值以后的结余。

```
def test(a, b, c = 3, d = 4, *args):
    print(locals())
```

```
>>> test(1, 2, 33)                    # 不足以填充普通参数和默认值参数
{
    'a': 1,
    'b': 2,
    'c': 33,
    'd': 4,
```

```
    'args': ()
}

>>> test(1, 2, 33, 44, 5, 6, 7)          # 填充完普通参数和默认值参数后，收集剩余的参数值
{
    'a': 1,
    'b': 2,
    'c': 33,
    'd': 44,
    'args': (5, 6, 7)
}
```

不能对收集参数命名传参。

```
def test(a, *args):
    pass
```

```
>>> test(a = 1, args = (2, 3))
TypeError: test() got an unexpected keyword argument 'args'
```

键值参数

常见的是在位置参数列表后放置一个键值收集参数。

```
def test(a, b, *args, **kwargs):
    print(kwargs)
```

```
>>> test(1, 2, 3, x = 1, y = 2)            # 多余的位置参数被 args 收集
{'x': 1, 'y': 2}

>>> test(b = 2, a = 1, x = 3, y = 4)       # kwargs 收集多余的键值参数
{'x': 3, 'y': 4}
```

键值收集仅针对命名传参，其对多余的位置参数没兴趣。

```
def test(a, b, **kwargs): pass
```

```
>>> test(1, 2, 3, x = 1)
TypeError: test() takes 2 positional arguments but 3 were given
```

在此基础上，Python 3 新增了一种名为 keyword-only 的键值参数类型。

1．以星号与位置参数列表分隔边界。

2．普通 keyword-only 参数，零到多个。

3．有默认值的 keyword-only 参数，零到多个。

4．双星号键值收集参数，仅一个。

无默认值的 keyword-only 必须显式命名传参，否则会被视为普通位置参数。

```
def test(a, b, *, c):
    print(locals())
```

```
>>> test(1, 2, 3)
TypeError: test() takes 2 positional arguments but 3 were given

>>> test(1, 2)
TypeError: test() missing 1 required keyword-only argument: 'c'
```

```
>>> test(1, 2, c = 3)
{'c': 3, 'b': 2, 'a': 1}
```

即便没有位置参数，keyword-only 也须按规则传递。

```
def test(*, c): pass
```

```
>>> test(1)
TypeError: test() takes 0 positional arguments but 1 was given

>>> test(c = 1)
```

除单星号外，位置收集参数（*args）也可作为边界。但只取其一，不能同时出现。

```
def test(a, *args, c, d = 99, **kwargs):
    print(locals())
```

```
>>> test(1, 2, 3, c = 88, x = 10, y = 20)
{
    'a': 1,
    'args': (2, 3),
    'c': 88,
    'd': 99,
    'kwargs': {'x': 10, 'y': 20},
}
```

同样不能对键值收集参数命名传参。否则，其结果是弄巧成拙，被当作普通参数收集。

```
def test(**kwargs):
    print(kwargs)
```

```
>>> test(kwargs = {"a":1, "b":2})      # 被当作普通键值参数收集，kwargs["kwargs"]
{'kwargs': {'a': 1, 'b': 2}}

>>> test(**{"a":1, "b":2})             # 这才是正确的姿势
{'a': 1, 'b': 2}
```

默认值

参数的默认值允许省略实参传值，这让函数调用更加灵活。尤其是那些参数众多，或具有缺省设定的函数。

默认值在函数对象创建时生成，保存在__defaults__，为每次调用所共享。

如此一来，其行为类似于静态局部变量，会“记住”以往的调用状态。

```
def test(a, x = [1, 2]):
    x.append(a)
    print(x)
```

```
>>> test.__defaults__
```

```
([1, 2],)
```

> 默认值对象作为函数构建参数而存在。

```
>>> dis.dis(compile("def test(a, x = [1, 2]): pass", "", "exec"))
  1           0 LOAD_CONST               0 (1)
              2 LOAD_CONST               1 (2)
              4 BUILD_LIST               2                         # 构建默认值对象
              6 BUILD_TUPLE              1                         # 作为构建参数
              8 LOAD_CONST               2 (<code object test>)
             10 LOAD_CONST               3 ('test')
             12 MAKE_FUNCTION            1                         # 参数 1 表示包含缺省参数
```

如默认值为可变类型，且在函数内做了修改，那么后续调用会观察到本次改动，这导致默认值失去原本的含义。

```
>>> test(3)
[1, 2, 3]

>>> test(4)                    # 在上次调用的基础上添加
[1, 2, 3, 4]
```

故建议默认值选用不可变类型，或以 None 表示可忽略。

```
def test(a, x = None):
    x = x or []                # 忽略时，主动新建
    x.append(a)
    return x
```

```
>>> test(1)
[1]

>>> test(2)
[2]

>>> test(3, [1, 2])           # 提供非默认值实参
[1, 2, 3]
```

其实静态局部变量还是有实际用处的，其可在不用外部变量的情况下维持函数状态。比如，用来存储调用计数等。但相比于参数默认值，正确的做法是为函数创建一个状态属性。毕竟变量为函数内部使用，而参数属于对外接口。所创建的属性与函数对象的生命周期相同，不会随调用结束而终结。

会变动的默认值，还会导致测试出现不一致性。

```
def test():
    test.__count__ = hasattr(test, "__count__") and test.__count__ + 1 or 1
    print(test.__count__)
```

```
>>> test()
1

>>> test()
2
```

形参赋值

解释器对形参赋值的过程如下。

1．按顺序对位置参数赋值。

2．按命名方式对指定参数赋值。

3．收集多余的位置参数。

4．收集多余的键值参数。

5．为没有赋值的参数设置默认值。

6．检查参数列表，确保非收集参数都已赋值。

收集参数 args、kwargs 属于习惯性命名，并非强制要求。

对应形参的顺序，实参也有一些基本规则。

- 无默认值参数，必须有实参传入。
- 键值参数总是以命名方式传入。
- 不能对同一参数重复传值。

无论是以位置、命名方式，还是星号展开，都不能对同一参数重复传值。

```
def test(a, b): pass
```

```
>>> test(1, 2, a = 1)
TypeError: test() got multiple values for argument 'a'

>>> test(**{"a":1, "b":2}, **{"a":1})
TypeError: test() got multiple values for keyword argument 'a'

>>> test(*(1, 2), **{"a":1})
TypeError: test() got multiple values for argument 'a'
```

> 键值收集参数会维持传入顺序（PEP468）。如键值参数存在次序依赖，那么此功能就有实际意义。另外，收集参数不计入__code__.co_argcount。

签名设计

设计一个显示函数，那么其理想的签名应该如下：

```
def printx(*objects, sep = ",", end = "\n"):
```

> 参数分作两部分：待显示对象为主，可省略的显示设置为次。从设计角度来讲，待显示对象是外部资源，而设置项用于控制函数自身行为，其分属不同范畴。这如同卡车所装载的货物，与车自身控制系统的差别。

据此接口，对象数量未定，其设置除默认值外，还可显式调整。如使用 Python 2，则作为收集参数的 objects 就只能放在参数列表尾部。

```
def printx(sep = ",", end = "\n", *objects):
    print(locals())
```

除主次不分导致不美观外，最大的麻烦是不能绕开默认设置，单独为 objects 传值。加上收集参数无法命名传参，这直接导致默认配置项毫无意义。

```
>>> printx(1, 2, 3)
{'objects': (3,), 'end': 2, 'sep': 1}

>>> printx(objects = (1, 2))
TypeError: printx() got an unexpected keyword argument 'objects'
```

而 Python 3 的 keyword-only 参数可完美地解决此问题。

```
def printx(*objects, sep = ",", end = "\n"):
    print(locals())
```

```
>>> printx(1, 2, 3)
{'objects': (1, 2, 3), 'end': '\n', 'sep': ','}

>>> printx(1, 2, 3, sep = "|")
{'objects': (1, 2, 3), 'end': '\n', 'sep': '|'}
```

参数列表还不宜过长，可尝试将部分参数重构为复合对象。

复合参数变化与函数分离。这样一来，添加字段，或修改缺省值，就不影响已有用户了。

4.3 返回值

函数没有返回值的定义，具体返回什么、返回几个，全凭“任性”。

```
def test(n):
    if n > 0:
        return 1, 2, 3
    elif n < 0:
        return -1, -2
```

```
    return 0
```

```
>>> test(1)
(1, 2, 3)

>>> test(-1)
(-1, -2)

>>> test(0)
0
```

这种返回数量忽多忽少的写法，在有明确返回值定义的语言里是决不允许的。

从实现角度看，只要返回数量多于一，编译器就将其打包成元组对象。如此一来，所谓忽多忽少，无非是单个元组里的元素数量多寡而已。

```
def test(a, b):
    return "hello", a + b
```

```
>>> dis.dis(test)
  2           0 LOAD_CONST               1 ('hello')
              2 LOAD_FAST                0 (a)
              4 LOAD_FAST                1 (b)
              6 BINARY_ADD
              8 BUILD_TUPLE              2             # 打包成元组，返回
             10 RETURN_VALUE
```

```
>>> x = test(1, 2)

>>> type(x)
<class 'tuple'>

>>> x
('hello', 3)
```

即便什么都不做，默认也返回 None。如此一来，函数总返回结果，可确保作为表达式使用。

```
>>> def test(): pass
```

```
>>> dis.dis(test)
  1           0 LOAD_CONST               0 (None)
              2 RETURN_VALUE
```

```
>>> def test(): return

>>> dis.dis(test)
  1           0 LOAD_CONST               0 (None)
              2 RETURN_VALUE
```

至于用多变量接收返回值，实际是对返回的元组进行序列解包。

```
>>> dis.dis(compile("a, b = test()", "", "exec"))
  1           0 LOAD_NAME                0 (test)
              2 CALL_FUNCTION            0
              4 UNPACK_SEQUENCE          2                # 解包
              6 STORE_NAME               1 (a)
              8 STORE_NAME               2 (b)
             10 LOAD_CONST               0 (None)
             12 RETURN_VALUE
```

4.4 作用域

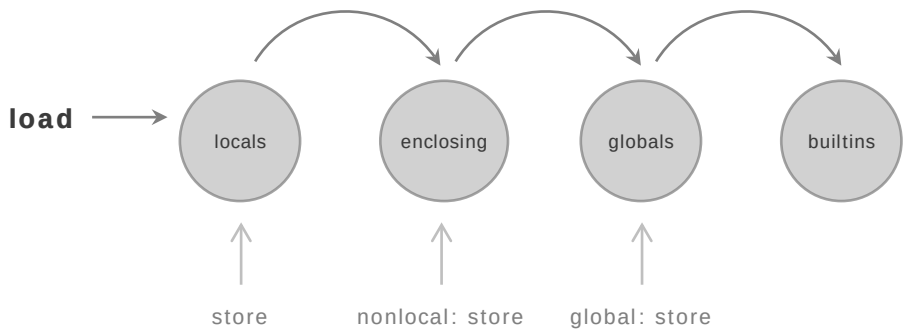
在函数内访问变量，会以特定顺序依次查找不同层次的作用域。

```
>>> builtins.B = "B"
>>> G = "G"

>>> def enclosing():
        E = "E"

        def test():
            L = "L"
            print(L, E, G, B)

        return test
```

```
>>> enclosing()()
L E G B
```

此规则被简称作 LEGB。

> 这与赋值语句默认针对当前名字空间，或用 global、nonlocal 关键字做外部声明完全不同。赋值操作的目标位置是明确的，即便是 enclosing，也需要在编译时静态绑定。而 LEGB 则在运行期动态地从多个位置按特定顺序查找。



内存结构

函数每次调用，都会新建栈帧（stack frame），用于局部变量和执行过程存储。等执行结束，栈帧内存被回收，同时释放相关对象。

```
def test():
   print(id(locals()))

>>> test()
4460820664

>>> test()
4458753840
```

是否栈帧内存就是 locals 名字空间？

以字典实现的名字空间，虽然灵活，但存在访问效率低下等问题。这对于使用频率较低的模块名字空间尚可，可对于有性能要求的函数调用，显然就是瓶颈所在。

为此，解释器划出专门的内存空间，用效率最快的数组替代字典。在函数指令执行前，先将包含参数在内的所有局部变量，以及要使用的外部变量复制（指针）到该数组。基于作用域不同，此内存区域可简单分作两部分：FAST 和 DEREF。

如此，操作指令只需用索引即可立即读取或存储目标对象，这远比哈希查找过程高效得多。从前面的反汇编开始，我们就看到了大量类似于 LOAD_FAST 的指令，其参数就是索引号。

```
def enclosing():

    E = "E"

    def test(a, b):
        c = a + b
        print(E, c)

    return test
```

```
>>> t = enclosing()                    # 返回 test 函数

>>> t.__code__.co_varnames             # 局部变量列表（含参数）。与索引号对应
('a', 'b', 'c')

>>> t.__code__.co_freevars             # 所引用的外部变量列表。与索引号对应
('E',)
```

```
>>> dis.dis(t)
  4           0 LOAD_FAST                0 (a)       # 从 FAST 区域，以索引号访问并载入
              2 LOAD_FAST                1 (b)
              4 BINARY_ADD
              6 STORE_FAST               2 (c)       # 将结果存入 FAST 区域

  5           8 LOAD_GLOBAL              0 (print)
             10 LOAD_DEREF               0 (E)       # 从 DEREF 区域，访问并载入外部变量
             12 LOAD_FAST                2 (c)
             14 CALL_FUNCTION            2
             16 POP_TOP
             18 LOAD_CONST               0 (None)
             20 RETURN_VALUE
```

FAST 和 DEREF 数组大小是统计参数和变量得来的，对应的索引值也在编译期确定。所以，不能在运行期扩张。前面曾提及，global 关键字可向全局名字空间新建名字，但 nonlocal 不允许。其原因就是 nonlocal 代表外层函数，无法动态向其 FAST 数组插入或追加新元素。

另外，LEGB 中的 E 已被保存到 DEREF 数组，相应的查找过程也被优化，无须费时费力去迭代调用堆栈。所以，LEGB 是针对源码的说法，而非内部实现。

名字空间

现在的问题是，为何 locals 函数返回的是字典类型？实际上，除非调用该函数，否则函数执行期间，根本不会创建所谓名字空间字典。也就是说，函数返回的字典是按需延迟创建，并从 FAST 区域复制相关信息得来的。

在了解了此背景后，类似于下面的问题就很好解释了。

```
def test():
    locals()["x"] = 100                 # 运行期通过名字空间新建变量
    print(x)                            # 编译此指令时，本地并没有 x 这个名字
```

```
>>> test()                              # 失败
NameError: name 'x' is not defined
```

```
>>> dis.dis(test)
  2           0 LOAD_CONST               1 (100)
```

```
              2 LOAD_GLOBAL              0 (locals)
              4 CALL_FUNCTION            0
              6 LOAD_CONST               2 ('x')
              8 STORE_SUBSCR

  3          10 LOAD_GLOBAL              1 (print)
             12 LOAD_GLOBAL              2 (x)          # 编译时确定，从全局而非 FAST 载入
             14 CALL_FUNCTION            1
             16 POP_TOP
             18 LOAD_CONST               0 (None)
             20 RETURN_VALUE
```

很显然，名字使用静态作用域。运行期行为，对此并无影响。而另一方面，所谓的 locals 名字空间不过是 FAST 的复制品，对其变更不会同步到 FAST 区域。

```
def test():
    x = 100
    locals()["x"] = 999                    # 新建字典，进行复制。对复制品的修改不会影响 FAST

    print("fast.x =", x)
    print("locals.x =", locals()["x"])     # 从 FAST 刷新，修改丢失
```

```
>>> test()
fast.x = 100
locals.x = 100
```

至于 globals 能新建全局变量，并影响外部环境，是因为模块直接以字典实现名字空间，没有类似于 FAST 的机制。

Python 2 可通过插入 exec 语句影响名字作用域的静态绑定，但对 Python 3 无效。

栈帧会缓存 locals 函数所返回的字典，以避免每次均新建。如此，可用它存储额外的数据，比如向后续逻辑提供上下文状态等。但请注意，只有再次调用 locals 函数，才会刷新字典数据。

```
def test():
    x = 1
```

```
    d = locals()
    print(d is locals())          # ① 每次返回同一字典对象

    d["context"] = "hello"        # ② 存储额外的数据
    print(d)

    x = 999                       # ③ 修改 FAST 时，不会主动刷新 locals 字典
    print(d)                      #    依旧输出上次调用 locals 的结果

    print(locals())               # ④ 刷新操作由 locals 调用触发
    print(d)
```

```
>>> test()
① True
② {'x': 1, 'context': 'hello'}
③ {'x': 1, 'context': 'hello'}
④ {'x': 999, 'context': 'hello'}
④ {'x': 999, 'context': 'hello'}
```

> 函数每次调用都会新建栈帧，也就是说 locals 所返回字典的生命周期仅对应本次调用。

静态作用域

在对待作用域这个问题上，编译器也许比你想象的要“笨”一些。

```
def test():
    if False: x = 100
    print(x)
```

```
>>> test()
UnboundLocalError: local variable 'x' referenced before assignment

>>> dis.dis(test)
  4           0 LOAD_GLOBAL              0 (print)
              2 LOAD_FAST                0 (x)            # x 作用域：FAST
              4 CALL_FUNCTION            1
```

虽然编译器将死代码剔除了，但其对 x 作用域的影响依然存在。

似乎也不能说编译器笨。既然“程序猿”在代码里写了赋值语句，那么在最初的设想里，x 显然是本地变量。至于最后死代码执行与否，就是另外一回事了。

```
def test():
    if False: global x
    x = 100
```

```
>>> dis.dis(test)
  4           0 LOAD_CONST               1 (100)
              2 STORE_GLOBAL             0 (x)          # x 作用域：GLOBAL
```

建议

函数最好设计为纯函数，仅依赖参数、内部变量和自身属性；依赖外部状态，会给重构和测试带来诸多麻烦。或许可将外部依赖变成 keyword-only 参数，如此测试就可自定义依赖环境，以确保最终结果一致。

如必须依赖外部变量，则尽可能不做修改，以返回值交由调用方决策。

纯函数（pure function）输出与输入以外的状态无关，没有任何隐式依赖。相同输入总是输出相同结果，且不对外部环境产生影响。

注意区分函数和方法的设计差异。函数以逻辑为核心，通过输入条件计算结果，尽可能避免持续状态。而方法则围绕实例状态，持续展示和连续修改。所以，方法与实例状态共同构成了封装边界，这与函数设计理念不同。

4.5 闭包

闭包是指函数离开生成环境后，依然可记住，并持续引用词法作用域里的外部变量。

```
def make():
    x = [1, 2]
    return lambda: print(x)
>>> a = make()
>>> a()
[1, 2]
```

> 如果不考虑闭包因素，这段代码有很大问题。因为 x 生命周期是 make 栈帧，调用结束后理应被销毁。LEGB 仅是执行期行为，对这个示例而言，匿名函数显然无法构成引用。但实际结果是，所返回的匿名函数依然可以访问 x 变量，这就是所谓闭包效应。

关于闭包，业界有很多学术解释。简单一点说，其就是函数和所引用环境变量的组合体。从这点上来说，闭包不等于函数，而只是形式上返回函数而已。因引用外部状态，闭包函数自然也不是纯函数。再加上闭包会延长环境变量的生命周期，我们理应慎重使用。

创建

既然闭包由两部分组成，那么其创建过程可分为：

1．打包环境变量。

2．将环境变量作为参数，新建要返回的函数对象。

> 因生命周期的改变，环境变量存储区从 FAST 转移到 DEREF。
>
> 闭包函数可以是匿名函数，也可以是普通函数。

```
def make():
    x = 100
    def test(): print(x)
    return test
```

```
>>> dis.dis(make)
  2           0 LOAD_CONST               1 (100)
              2 STORE_DEREF              0 (x)                    # 保存在 DEREF
```

```
  3           4 LOAD_CLOSURE             0 (x)                       # 闭包环境变量
              6 BUILD_TUPLE              1
              8 LOAD_CONST               2 (<code object test">)
             10 LOAD_CONST               3 ('make.<locals>.test')
             12 MAKE_FUNCTION            8                           # 创建函数时包含闭包参数
             14 STORE_FAST               0 (test)

  4          16 LOAD_FAST                0 (test)
             18 RETURN_VALUE
```

同样，闭包函数也得从 DEREF 读取环境变量。

```
>>> f = make()
>>> dis.dis(f)
  3           0 LOAD_GLOBAL              0 (print)
              2 LOAD_DEREF               0 (x)              # 从 DEREF 载入闭包环境变量
              4 CALL_FUNCTION            1
              6 POP_TOP
              8 LOAD_CONST               0 (None)
             10 RETURN_VALUE
```

自由变量

闭包所引用的环境变量也被称作自由变量，它被保存在函数对象的__closure__属性中。

```
def make():
    x = [1, 2]
    print(hex(id(x)))
    return lambda: print(x)
```

```
>>> f = make()
0x10eeae108

>>> f.__closure__
(<cell at 0x10ed47198: list object at 0x10eeae108>,)

>>> f()
[1, 2]
```

函数所引用的自由变量名，可通过代码对象的相关属性获取。

```
>>> f.__code__.co_freevars              # 当前函数引用外部自由变量列表
('x',)

>>> make.__code__.co_cellvars           # 被内部闭包函数引用的变量列表
('x',)
```

自由变量保存在函数对象里，那多次调用是否被覆盖？自然不会，因为所返回的函数对象也是每次新建的。要知道，创建闭包等于“新建函数对象，附加自由变量”。

```
def make(x):
    return lambda: print(x)
```

```
>>> a = make([1, 2])
>>> b = make(100)

>>> a is b                                          # 每次返回新的函数对象实例
False

>>> a.__closure__                                   # 自由变量保存在各自独立的函数实例内
(<cell at 0x10ed61c78: list object at 0x10ee7fcc8>,)

>>> b.__closure__
(<cell at 0x10ed47d68: int object at 0x10d0c6970>,)
```

多个闭包函数可共享同一自由变量。

我们尝试创建共享数据池（data）的队列操作函数。

```
def queue():
    data = []

    push = lambda x: data.append(x)
    pop = lambda: data.pop(0) if data else None

    return push, pop
```

```
>>> push, pop = queue()
```

```
>>> push.__closure__
(<cell at 0x10ed612e8: list object at 0x10edcae48>,)       # 共享自由变量

>>> pop.__closure__
(<cell at 0x10ed612e8: list object at 0x10edcae48>,)
```

```
>>> for i in range(10, 13):
        push(i)

>>> while True:
        x = pop()
        if not x: break
        print(x)

10
11
12
```

闭包让函数持有状态，其可部分实现 class 功能。但这应局限于特定的小范围，避免隐式状态依赖对代码测试、阅读和维护造成麻烦。

自引用

在函数内引用函数自己，也可构成闭包。

当 def 创建函数对象后，会在当前名字空间将其与函数名字关联。如此，该函数实例自然也可作为自由变量。

```
def make(x):

    def test():
        test.x = x              # 引用自己
        print(test.x)           # 引用的是当前函数实例，其效果类似于 this

    return test
```

```
>>> a, b = make(1234), make([1, 2])
```

```
>>> a.__closure__
(<cell at 0x10ed61798: function object at 0x10ed50ea0>,      # 在自由变量列表中包含自己
 <cell at 0x10ed61318: int object at 0x10ee10890>)

>>> b.__closure__
(<cell at 0x10ed61618: function object at 0x10efb2840>,
 <cell at 0x10ed61918: list object at 0x10ee6d848>)
```

在函数内引用自己时，自然是本次新建的函数实例，无须担心数据共享。

```
>>> dis.dis(a)
  3           0 LOAD_DEREF               1 (x)
              2 LOAD_DEREF               0 (test)
              4 STORE_ATTR               0 (x)

  4           6 LOAD_GLOBAL              1 (print)
              8 LOAD_DEREF               0 (test)
             10 LOAD_ATTR                0 (x)
             12 CALL_FUNCTION            1
             14 POP_TOP
             16 LOAD_CONST               0 (None)
             18 RETURN_VALUE
```

```
>>> a()
1234

>>> b()
[1, 2]
```

延迟绑定

闭包只是绑定自由变量，并不会立即计算引用内容。只有当闭包函数执行时，才访问所引用的目标对象。这样就有所谓的延迟绑定（late binding）现象。

```
def make(n):
    x = []

    for i in range(n):
```

```
        x.append(lambda: print(i))

    return x
```

```
>>> a, b, c = make(3)
>>> a(), b(), c()
2
2
2
```

```
>>> a.__closure__
(<cell at 0x10efaf798: int object at 0x10d0c5d30>,)

>>> b.__closure__
(<cell at 0x10efaf798: int object at 0x10d0c5d30>,)

>>> c.__closure__
(<cell at 0x10efaf798: int object at 0x10d0c5d30>,)
```

输出结果不是 0, 1, 2？

整理一下执行次序：

1. make 创建并返回 3 个闭包函数，引用同一自由变量 i。
2. make 执行结束，i 等于 2。
3. 执行闭包函数，引用并输出 i 的值，自然都是 2。

就这么简单？

从__closure__来看，函数并不直接存储自由变量，而是 cell 包装对象，以此间接引用目标。每个自由变量都被打包成一个 cell。循环期间虽然 cell 也和 i 一样引用不同整数对象，但这对尚未执行的闭包函数没有影响。循环结束，cell 引用目标确定下来，这才是闭包函数执行时的输出结果。

根据其他语言的使用经验，解决延迟绑定问题，要么立即复制，要么引用不同自由变量。但改成复制后，结果并不如人意。

```
def make(n):
    x = []
    for i in range(n):
        c = i                                # 复制对象? 并不是，这是一个坑
        x.append(lambda: print(c))

    return x
```

```
>>> a, b, c = make(3)
>>> a(), b(), c()
2
2
2
```

这里未能得到预期结果。原因并不复杂，变量 c 的作用域是函数，而非 for 语句。也就是说，不管执行多少次循环，也仅有一个 c 存在。如此一来，闭包函数依然绑定同一自由变量，这与复制目标对象无关。这是不同语言作用域规则不同而导致的经验错误。

```
>>> a.__closure__
(<cell at 0x10ed47d38: int object at 0x10d0c5d30>,)        # 仍是同一对象

>>> b.__closure__
(<cell at 0x10ed47d38: int object at 0x10d0c5d30>,)

>>> c.__closure__
(<cell at 0x10ed47d38: int object at 0x10d0c5d30>,)
```

顺此思路，可换一个坚决不共享的变量，比如在函数创建时就立即计算的参数默认值。

```
def make(n):
    x = []
    for i in range(n):
        x.append(lambda o = i: print(o))

    return x
```

```
>>> a, b, c = make(3)
>>> a(), b(), c()
0
1
2
```

> 匿名函数参数 o 是私有变量，设置默认值时，其复制当前的 i 引用。虽然只有一个变量 i，但在循环过程中，它指向不同的整数对象。所以，复制的引用也就不同。
>
> 最重要的是，这样就没有闭包了。

```
>>> a.__closure__
>>> b.__closure__
>>> c.__closure__

>>> make.__code__.co_cellvars
()
```

优缺点

闭包的缺点可能和优点一样明显。

闭包具备封装特征，可实现隐式上下文状态，并减少参数。在设计上，其可部分替代全局变量，或将执行环境与调用接口分离。

其首要缺点是，对自由变量隐式依赖，会提升代码的复杂度，这直接影响测试和维护。其次，自由变量生命周期的提升，会提高内存占用。

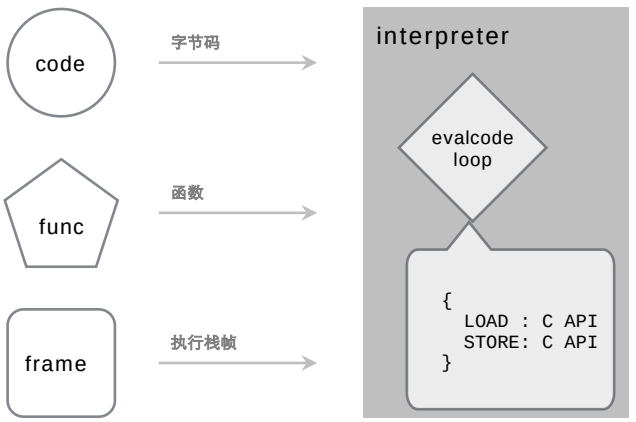
> 应控制隐式依赖的范围和规模，能省则省。

4.6 调用

在前面的不同章节里，多次提及字节码解释执行。那么当函数被调用时，具体情形如何？

整个系统核心是被称为解释器（interpreter）的组件。其从实现代码来看很疯狂：重度肥胖函数，外加超大循环。让我们抛开技术名词，用例子说明其中的关系。

假设解释器是一台 ATM 取款机。当储户发出“取款”指令（字节码）时，机器触发预置功能列表中与之对应的操作，以银行卡为参数，检查并修改账户数据，然后出钞。所谓指令不过是内部某个功能的“名字”而已，其仅作为选择条件，并不参与机器执行。



在解释器内部，每条字节码指令对应一个完全由 C 实现的逻辑。

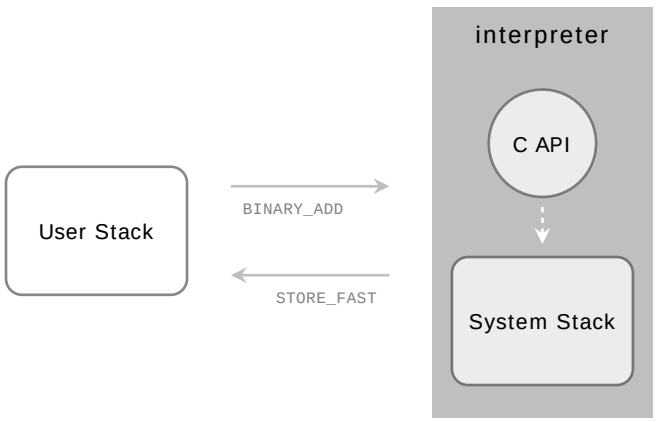
解释器运行在系统线程上，那如何处理内部系统代码和用户代码数据呢？从反汇编结果来看，就算字节码指令被解释为内部调用，可依然有参数和返回值需要存储。

继续以上面的例子解释，这里实际有两个存储空间，机器内部（系统栈）和储户钱包（用户栈）。取款时，银行卡从钱包传递到机器，最后连同钞票放回钱包。在操作完成后，机器准备下次交易，本次数据被清除。与用户相关的数据都在钱包内。所以说，系统栈用于机器执行，用户栈存储用户代码执行状态。

当函数被调用时，会专门为其分配用户栈内存。用户栈内存除用来存储变量外，还包括字节码参数和返回值所需的空间。对系统指令来说，这里只能存放用户指令数据。如此一来，两方各有所属，确保数据互不影响。

```
def add(a, b):
    c = a + b
    return c
```

```
>>> dis.dis(add)
  2           0 LOAD_FAST                0 (a)      # 从 FAST 读取参数 a，压入用户栈
              2 LOAD_FAST                1 (b)      # 从 FAST 读取参数 b，压入用户栈
              4 BINARY_ADD                          # 系统指令从用户栈读取操作数，执行加法操作
              6 STORE_FAST               2 (c)      # 将结果写回 FAST

  3           8 LOAD_FAST                2 (c)
             10 RETURN_VALUE
```



最后，编译器没有函数内联，没有深度优化。即便是空函数，依旧生成字节码指令，需要解释器执行。

> 解释器参数 -O 和 -OO，并非指令优化开关，而是为了移除调试代码和帮助文档。

```
>>> code ="""
    def test(): pass
    test()
"""

>>> dis.dis(compile(code, "", "exec", optimize = 2))          # 启用-OO 优化方式
  2           0 LOAD_CONST               0 (<code object>)
              2 LOAD_CONST               1 ('test')
              4 MAKE_FUNCTION            0                   # 创建函数
              6 STORE_NAME               0 (test)

  3           8 LOAD_NAME                0 (test)
             10 CALL_FUNCTION            0                   # 调用函数
```

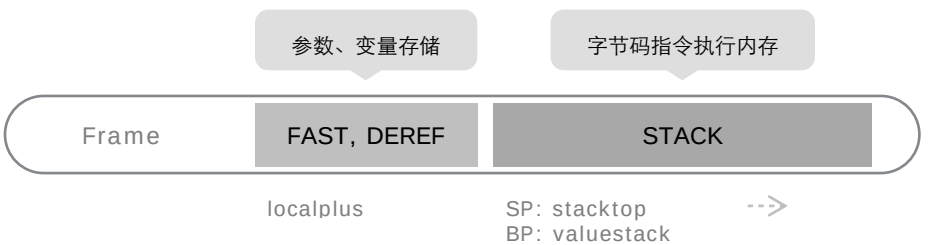
```
             12 POP_TOP
             14 LOAD_CONST               2 (None)
             16 RETURN_VALUE
```

调用堆栈

我们通常将进程内存分作堆（heap）和栈（stack）两类：堆可自由申请，通过指针存储自由数据；而栈则用于指令执行，与线程相绑定。函数调用和执行都依赖线程栈存储上下文和执行状态。

在函数 A 内调用函数 B，须确保 B 结束后能回转到 A，并继续执行后续指令。这就要求将 A 的后续指令地址预先存储起来。调用堆栈（call stack）的基本用途便是如此。

除返回地址外，还须为函数提供参数、局部变量存储空间。依不同调用约定，甚至要为被调用函数提供参数和返回值内存。显然，在线程栈这块内存里，每个被调用函数都划有一块保留地，我们将其称作栈帧（stack frame）。



> 基于栈式虚拟机（Stack-based VM）设计的字节码指令集，没有寄存器的相关概念，内部使用变量实现类似于 SP、BP、PC/IP 的寄存器功能。

因解释执行的缘故，字节码指令数据使用独立的用户栈空间。且与系统栈连续内存不同，用户栈帧由独立对象实现，以链表形式构成完整的调用堆栈。其好处是不受系统栈大小的制约，缺点是性能方面要差一些。但考虑到它只存储数据，实际执行过程依然在系统栈完成，这倒也能被接受。

因栈帧使用频繁，系统会缓存 200 个栈帧对象，并按实际所需调整内存大小。

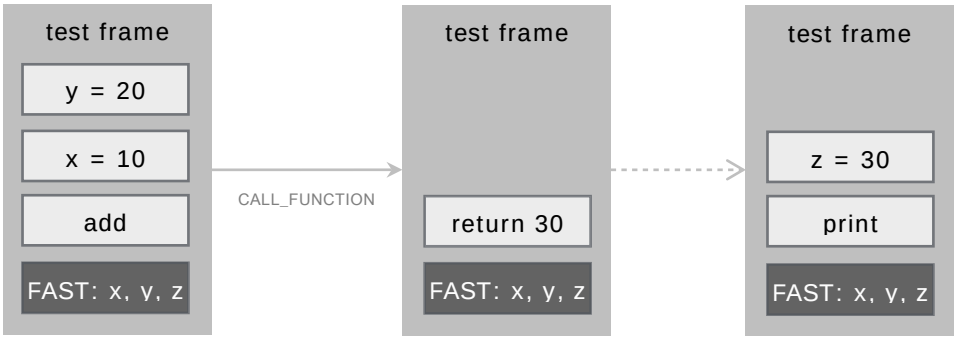
操作系统对线程栈大小的限制可使用 ulimit -s 查看，最新 64 位系统通常为 8 MB。

一旦函数执行（比如递归）内存超出限制，就会引发堆栈溢出（stack overflow）错误。

在栈帧内，变量内存通常固定不变，执行内存视具体指令重复使用。

```
def add(x, y):
    return x + y

def test():
    x = 10
    y = 20
    z = add(x, y)
    print(z)
```

```
>>> dis.dis(test)
  2           0 LOAD_CONST               1 (10)
              2 STORE_FAST               0 (x)

  3           4 LOAD_CONST               2 (20)
              6 STORE_FAST               1 (y)

  4           8 LOAD_GLOBAL              0 (add)       # 将待调用函数 add 入栈
             10 LOAD_FAST                0 (x)         # 将变量 x 入栈
             12 LOAD_FAST                1 (y)         # 将变量 y 入栈
             14 CALL_FUNCTION            2             # 调用函数（2 为参数数量）
             16 STORE_FAST               2 (z)         # 将返回值从栈保存到变量区

  5          18 LOAD_GLOBAL              1 (print)
             20 LOAD_FAST                2 (z)
             22 CALL_FUNCTION            1
             24 POP_TOP                                # 清除 print 返回值，确保栈平衡
             26 LOAD_CONST               0 (None)
             28 RETURN_VALUE
```



调用堆栈常出现在调试工具中，用于检视调用过程，以及各级环境变量取值。当然，也可在代码中使用，比如获取上级函数设置的上下文信息。

> 函数 sys._getframe 可访问调用堆栈内不同层级的栈帧对象。参数 0 为当前函数，1 为上级函数。

```
def A():
    x = "func A"
    B()

def B():
    C()

def C():
    f = sys._getframe(2)        # 向上 2 级，获取 A 栈帧
    print(f.f_code)             # A 代码对象
    print(f.f_locals)           # A 名字空间（此时为运行期）
    print(f.f_lasti)            # A 最后执行指令偏移量（以确定继续执行位置）
```

```
>>> A()
<code object A>
{'x': 'func A'}
6
```

```
>>> dis.dis(A)
  2           0 LOAD_CONST               1 ('func A')
              2 STORE_FAST               0 (x)
```

```
3           4 LOAD_GLOBAL              0 (B)
            6 CALL_FUNCTION            0                  <------ A.lasti
            8 POP_TOP
           10 LOAD_CONST               0 (None)
           12 RETURN_VALUE
```

请注意，无论是在函数内调用 globals，还是通过 frame.f_globals 访问，总返回定义该函数的模块名字空间，而非调用处。

另有 sys._current_frame 返回所有线程的当前栈帧，用来确定解释器的工作状态。只是文档里这两个函数都标记为内部使用。可用标准库 inspect 进行替代，它拥有更多操作函数。

如果只是输出调用过程，可使用 traceback 模块，这类似于解释器输出错误信息。

```
import inspect

def A(): B()
def B(): C()

def C():
    for f in inspect.stack():
        print(f.function, f.lineno)

A()
```

输出：

```
$ python test.py

C 7
B 4
A 3
<module> 10
```

递归

递归深度受限，可使用 sys.getrecursionlimit、setrecursionlimit 查看和调整。

test.py

```
import sys

sys.setrecursionlimit(50)

def test():
    test()                    # 递归调用

test()
```

输出：

```
$ python test.py
RecursionError: maximum recursion depth exceeded
```

递归常被用来改善循环操作，比如树状结构遍历。当然，它须承担函数调用的额外开销，类似栈帧创建等。在不支持尾递归优化的情况下，这种负担尤为突出。

比如，函数 A 的最后动作是调用 B，并直接返回 B 的结果。那么 A 的栈帧状态就无须保留，其内存可直接被 B 覆盖使用。另外，将函数调用优化成跳转指令，可以大大提升执行性能。如此方式，被称作尾调用消除或尾调用优化（Tail Call Optimization，TCO）。

如果 A 尾调用自身，那么就成了尾递归。鉴于重复使用同一栈帧内存，这可避免堆栈溢出。不过 CPython 因实现方式的问题，对此并不支持。

包装

对已有函数，可通过包装形式改变其参数列表，使其符合特定调用接口。

```
def test(a, b, c):
    print(locals())
```

```
>>> import functools

>>> t = functools.partial(test, b = 2, c = 3)

>>> t(1)
{'c': 3, 'b': 2, 'a': 1}
```

原理很简单，包装函数将相关参数合并后，再调用原目标即可。

示意伪码

```
def partial(func, *part_args, **part_kwargs):

    def wrap(*call_args, **call_kwargs):
        kwargs = part_kwargs.copy()                        # 复制包装键值参数
        kwargs.update(call_kwargs)                         # 使用调用键值参数更新包装键值参数
        return func(*part_args, *call_args, **kwargs)      # 按顺序展开参数

    return wrap
```

基本合并规则：

1．包装位置参数优先。

2．调用键值参数覆盖包装键值参数。

3．合并后不能对单个目标参数多次赋值。

```
>>> functools.partial(test, 1, 2)(3)                 # 包装位置参数优先
{'a': 1, 'b': 2, 'c': 3}

>>> functools.partial(test, 1, c = 3)(2, c = 100)     # 调用键值参数覆盖包装参数
{'a': 1, 'b': 2, 'c': 100}
```

包装函数内存储了相关信息，可供查阅。

```
> t = functools.partial(test, 1, c = 3)

>>> t.func                                          # 原目标函数
<function __main__.test>

>>> t.args                                          # 包装位置参数
(1,)

>>> t.keywords                                      # 包装键值参数
{'c': 3}
```

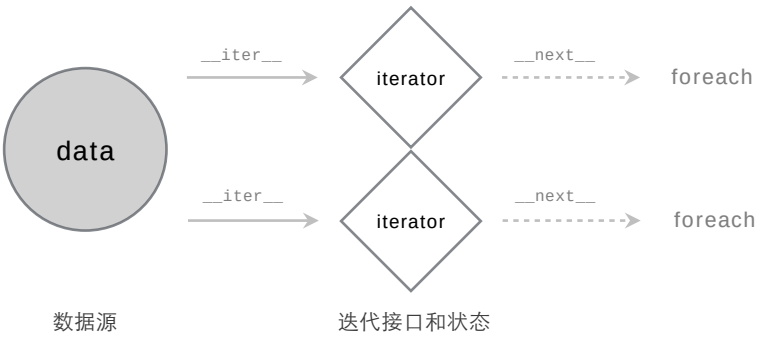
第 5 章 迭代器

5.1 迭代器概述

迭代是指重复从对象中获取数据，直至结束。而所谓迭代协议，概括起来就是用__iter__方法返回一个实现了__next__方法的迭代器对象。

> 实现__iter__方法，表示目标为可迭代（iterable）类型，允许执行手动或自动迭代操作。该方法新建并返回一个迭代器（iterator）实例。随后，通过调用 iterator__next__依次返回结果，直至抛出 StopIteration 异常表示结束。

我们注意到，迭代器使用了分离设计。首先，对目标对象而言，迭代器只是一种与自身逻辑无关的用户接口，组合显然比内联更合适；其次，迭代分多次完成，需要保存进度。而且还可能有重复迭代，以及同时进行多个迭代等情形。如何存储并管理这些状态？不如每次按需新建实例，单向存储进度。用完即毁，无须考虑太多。



> 可迭代类型未必就是序列数据，其也可能是按前后顺序操作的栈、队列，以及随机返回键值的哈希表，甚至是未知终点的网络数据流。

内置容器类型，还有一些常用函数均实现了迭代接口。

```
>>> issubclass(list, collections.Iterable)
True

>>> isinstance(range(3), collections.Iterable)
True

>>> isinstance(zip([1, 2]), collections.Iterable)
True
```

自定义类型

以 Python“包办家长”的性格，肯定有便捷方式创建迭代器类型。但为更好地理解协议流程和操作方式，在此先用笨方法逐一实现。

```
class Data:

    def __init__(self, n):
        self.data = list(range(n))

    def __iter__(self):
        return DataIter(self.data)                    # 返回新迭代器实例
```

```
class DataIter:

    def __init__(self, data):
        self.data = data
        self.index = 0

    def __next__(self):
        if not self.data or self.index >= len(self.data):
            raise StopIteration                              # 抛出异常表示迭代结束

        d = self.data[self.index]                            # 本次迭代返回数据
        self.index += 1                                      # 存储迭代进度
        return d
```

手工迭代

```
>>> d = Data(2)
>>> x = d.__iter__()

>>> x.__next__()
0

>>> x.__next__()
1

>>> x.__next__()
StopIteration
```

自动迭代

```
>>> for i in Data(2): print(i)
0
1
```

> 因无法通过__next__方法传入参数，自然也就无从重置状态或调整进度。迭代器实例本质上属于一次性消费，这也是每次在__iter__里新建的理由。

按理说，迭代器本身也应该是可迭代类型。但因其缺少__iter__实现，却无法使用。

```
>>> x = Data(2).__iter__()
```

```
>>> x.__iter__()
AttributeError: 'DataIter' object has no attribute '__iter__'
```

如此，可为其添加__iter__方法，用于返回自身。

```
class DataIter:

    def __iter__(self):
        return self

    ...
```

辅助函数

前面的示例有些笨拙，因为其内在列表容器已经实现迭代接口，直接返回便可，无须画蛇添足。当然，按照名字约定，我们不应直接调用__iter__方法，而是改用 iter 函数。

```
class Data:

    def __init__(self, n):
        self.data = list(range(n))

    def __iter__(self):
        return iter(self.data)
```

```
>>> Data(2).__iter__()
<list_iterator at 0x106288b00>
```

作为辅助函数，iter 还可为序列对象（__getitem__）自动创建迭代器包装。

```
class Data:

    def __init__(self, n):
        self.n = n

    def __getitem__(self, index):
        if index < 0 or index >= self.n: raise IndexError
        return index + 100
```

```
>>> iter(Data(2))
<iterator at 0x1062a82e8>
```

iter 甚至还对函数、方法等可调用类型（callable）进行包装。

> iter 可用于网络和文件等 I/O 数据接收，比起循环语句更优雅一些。

```
>>> x = lambda: input("n : ")                # 被__next__调用，无参数

>>> for i in iter(x, "end"): print(i)        # 函数 x 的返回值等于 end 时结束

n : 1
1
n : 2
2
n : end
```

与__next__方法对应的则是 next 函数，用于手动迭代。

```
>>> x = iter([1, 2])

>>> while True:
        try:
            print(next(x))
        except StopIteration:
            break

1
2
```

自动迭代

对于 for 循环语句，编译器会生成迭代相关指令，以实现对协议方法的调用。

```
def test():
    for i in [1, 2]: print(i)
```

```
>>> dis.dis(test)
              2 LOAD_CONST               3 ((1, 2))
```

```
              4 GET_ITER                          # 调用__iter__返回迭代器对象（或包装）
        >>    6 FOR_ITER                 12 (to 20)  # 调用__next__返回数据，结束则跳转
              8 STORE_FAST                0 (i)
             10 LOAD_GLOBAL               0 (print)
             12 LOAD_FAST                 0 (i)
             14 CALL_FUNCTION             1
             16 POP_TOP
             18 JUMP_ABSOLUTE             6          # 继续迭代
        >>   20 POP_BLOCK                            # 迭代结束
```

从解释器内部实现来看，GET_ITER 指令行为与 iter 函数类似。如目标对象实现了__iter__方法，则直接调用，否则尝试创建序列迭代包装。

设计意图

尽管列表、字典等容器类型实现了迭代器协议。但本质上，两者不属于同一层面。迭代器不仅是一种数据读取方法，而更多的是一种设计模式。

容器的核心是存储，其围绕数据提供操作方法，这是与用户逻辑无关的开放类型。而迭代器的重点是逻辑控制。调用方发出请求，随后决策由迭代器决定。数据内敛，抽象和实现分离。

5.2 生成器

生成器（generator）是迭代器的进化版本，用函数和表达式替代接口方法。其不仅简化了编码过程，还提供了更多控制能力用于复杂设计。

生成器函数的特殊之处在于，其内部以 yield 返回迭代数据。这与普通函数不同，无论内部逻辑如何，其函数调用总是返回生成器对象。随后，以普通迭代器方式继续操作。

```
def test():
    yield 1
    yield 2
```

```
>>> test()                                  # 函数调用返回生成器对象
<generator object test at 0x1060cbf68>
```

```
>>> for i in test(): print(i)               # 以普通方式迭代数据
1
2
```

每条 yield 语句对应一次__next__调用。可分列多条，或出现在循环语句中。只要结束函数流程，就相当于抛出迭代终止异常。

```
def test():
    for i in range(10):
        yield i + 100
        if i >= 1: return
```

```
>>> x = test()

>>> next(x)
100

>>> next(x)
101

>>> next(x)
StopIteration
```

子迭代器

如果数据源本身就是可迭代对象，那么可使用 yield from 子迭代器语句。和在 for 循环内用 yield 并无不同，只是语法更加简练。

```
def test():
    yield from "ab"
    yield from range(3)
```

```
>>> for o in test(): print(o)
a
b
```

```
0
1
2
```

生成器表达式

至于生成器表达式，除使用小括号外，其他规则与推导式完全相同。

```
>>> x = (i + 100 for i in range(8) if i % 2 == 0)

>>> x
<generator object <genexpr> at 0x1062f5990>
```

生成器表达式可被用作函数调用参数。

> 如果生成器表达式不是函数的唯一参数，为避免语法错误，就不能省略小括号。

```
def test(x):
    print(x)
    for i in x: print(i)
```

```
>>> test(i for i in range(3))
<generator object <genexpr> at 0x105efff68>
0
1
2
```

执行

相比于普通迭代器，生成器的执行过程稍显复杂。

首先，编译器会为生成器函数添加标记。对此类函数，解释器并不直接执行。而是将栈帧和代码作为参数，创建生成器实例。

```
def test(n):
    print("gen.start")
```

```
    for i in range(n):
        print(f"gen.yield {i}")
        yield i
        print("gen.resume")
```

```
>>> test.__code__.co_flags                    # generator
99

>>> inspect.isgeneratorfunction(test)
True
```

简单一点说，就是以通用生成器类型为模板，实现外部迭代器协议。在内部通过栈帧和代码参数，实现对用户代码的调用。

```
>>> x = test(2)

>>> x.gi_frame.f_locals                       # 栈帧内存储的函数调用参数
{'n': 2}

>>> x.gi_code                                 # 关联用户函数
<code object test>
```

```
>>> x.__next__                                # 实现迭代器协议方法
<method-wrapper '__next__' of generator>
```

所谓函数调用，不过是错觉。这也算解释执行的好处，起码不用插入额外的汇编代码。接下来，生成器对象在第一次__next__调用时触发，进入用户函数执行。

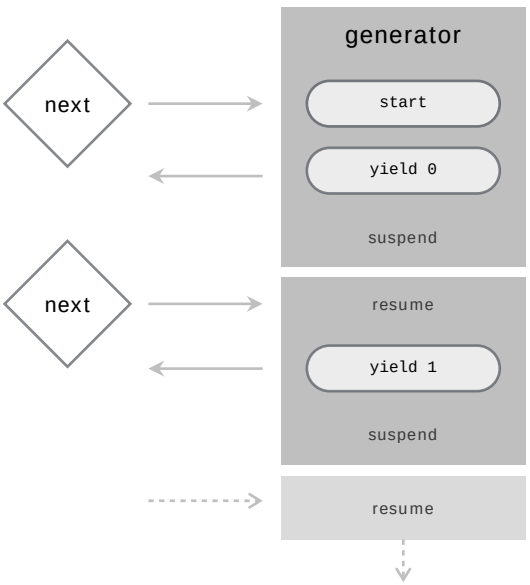
```
>>> next(x)
gen.start
gen.yield 0
0
```

当执行到 yield 指令时，在设置好返回值后，解释器保存线程状态，并挂起当前函数流程。只有再次调用__next__方法时，才能恢复状态，继续执行。如此，以 yield 为切换分界线，往复交替，直到函数结束。

执行状态保存在用户栈帧内，系统线程算是无状态多路复用，切换操作自然很简单。

```
>>> next(x)
gen.resume
gen.yield 1
1
```

```
>>> next(x)
gen.resume
StopIteration
```



方法

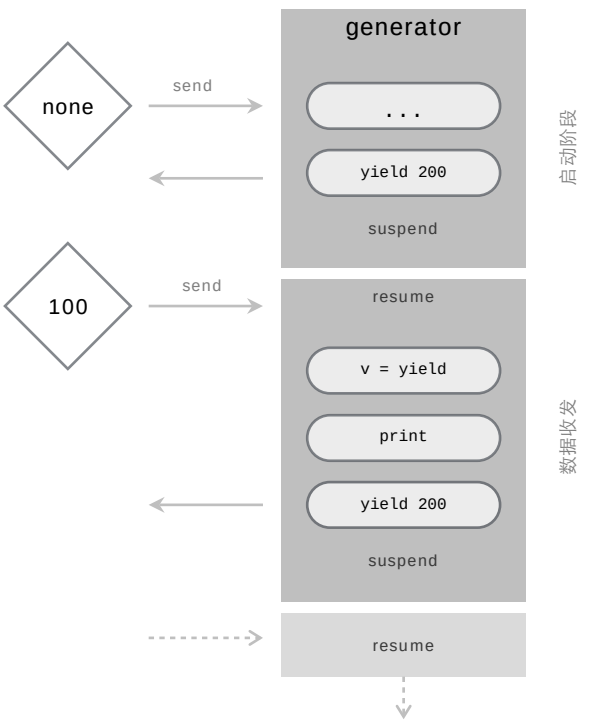
生成器的另一进化特征，就是提供双向通信能力。生成器不再是简单的数据提供方，其还可作为接收方存在。生成器甚至能在外部停止迭代，或发送信号实现重置等自定义行为。

方法 send 除可向 yield 传送数据外，其他与 next 完全一致。不过发送之前，须确保生成器已经启动。因为只有如此，才会进入函数执行流程，才会对外提供数据和交互。

```
def test():
    while True:
        v = yield 200
        print(f"resume {v}")
```

```
>>> x = test()

>>> x.send(None)              # 必须使用 next 或 send(None)启动生成器
200
```

```
>>> x.send(100)               # 可发送任何数据，包括 None，这与启动参数无关
resume 100
200
```



对生成器函数而言，挂起点是一个安全位置，相关状态被临时冻结。至于交互方式，只要将要发送的数据，或者其他状态标记放置到栈帧的指定位置，随后由解释器决定如何处理即可。

调用 close 方法，解释器将终止生成器迭代。

> 该方法在生成器函数内部引发 GeneratorExit 异常，通知解释器结束执行。
>
> 此异常无法捕获，但不影响 finally 的执行。

```
def test():
    for i in range(10):
        try:
            yield i
        finally:
            print("finally")
```

```
>>> x = test()

>>> next(x)                          # 启动生成器
0

>>> x.close()                        # 终止生成器
finally

>>> next(x)                          # 已经终止
StopIteration
```

还可像 send 发送数据那样，向生成器 throw 指定异常作为信号。

```
class ExitException(Exception): pass
class ResetException(Exception): pass


def test():
    while True:
        try:
            v = yield
            print(f"recv: {v}")
        except ResetException:
            print("reset.")
        except ExitException:
            print("exit.")
            return
```

```
>>> x = test()

>>> x.send(None)                    # 启动生成器
```

```
>>> x.throw(ResetException)         # 发出重置信号
reset.

>>> x.send(1)                       # 可继续发送数据
recv: 1
```

```
>>> x.throw(ExitException)          # 发出终止信号
exit.

>>> x.send(2)                       # 生成器已终止
StopIteration
```

> 异常属于合理流程控制，不能和错误完全等同起来。

5.3 模式

借助生成器切换执行功能，改善程序结构设计。

生产消费模型

在不借助并发框架的情况下，实现生产、消费协作。

> 消费者启动后，使用 yield 将执行权限交给生产者，等待其发送数据后激活处理。
>
> 如果有多个消费者，或数据处理时间较长，依然建议使用专业并发方案。

```
def consumer():
    while True:
        v = yield
        print(f"consume: {v}")
```

```
def producer(c):
    for i in range(10, 13):
        c.send(i)
```

```
c = consumer()                  # 创建消费者
c.send(None)                    # 启动消费者

producer(c)                     # 生产者发送数据
c.close()                       # 关闭消费者
```

输出：

```
consume: 10
consume: 11
consume: 12
```

消除回调

回调函数（callback）是常见异步接口设计方式。调用者在发起请求后，不再阻塞等待结果返回，而改由异步服务调用预先注册的函数来完成后续处理。

> 尽管回调函数使用广泛，但不太招人喜欢，甚至有“callback hell”的说法。究其原因，主要是回调方式让代码和逻辑碎片化，不利于阅读和维护。

设计一个简单异步服务示例，然后尝试用生成器清除回调函数。

```
def target(request, callback):
    s = time.time()
    request()                                           # 调用请求函数
    time.sleep(2)                                       # 模拟阻塞
    callback(f"done: {time.time() - s}")                # 调用回调函数，传入结果


def service(request, callback):
    threading.Thread(target=target, args=(request, callback)).start()
```

```
def request():                                          # 任务请求模拟
    print("start")

def callback(x):                                        # 任务结束通知
    print(x)

service(request, callback)
```

输出：

```
start
done: 2.003718137741089
```

首先，以生成器函数替代原来分离的两个函数。用 yield 分隔请求和返回代码，以便服务可介入其中。

```
def request():
    print("start")                                      # 请求部分

    x = yield                                           # 接收服务返回结果
    print(x)
```

接下来，改造服务框架，以生成器方式调用任务。

```
def target(fn):
    try:
        s = time.time()

        g = fn()                                        # 调用目标生成器函数
        g.send(None)                                    # 启动，开始执行请求代码

        time.sleep(2)
        g.send(f"done: {time.time() - s}")              # 阻塞结束后，将结果传回任务
    except StopIteration:                               # 目标结束，拦截异常
        pass


def service(fn):
    threading.Thread(target=target, args=(fn,)).start()
```

如此，既没有影响异步执行，也不再碎片化。

```
service(request)
print("do something")
```

输出：

```
start
do something                                          # 发起请求后，并未阻塞
done: 2.0059099197387695
```

协程

协程（coroutine）以协作调度方式，在单个线程上切换执行并发任务。配合异步接口，可将 I/O 阻塞时间用来执行更多任务。由于其在用户空间实现，因此也被称作用户线程。

理论上，任何具备 yield 功能的语言都能轻松实现协程。

当然，基于性能考虑，专业异步框架会选用 greenlet 之类的协程实现。

```
def sched(*tasks):
    tasks = list(map(lambda t: t(), tasks))      # 调用所有任务函数，生成器列表
    while tasks:                                 # 循环调用任务
        try:
            t = tasks.pop(0)                     # 从列表头部弹出任务
            t.send(None)                         # 开始执行该任务

            tasks.append(t)                      # 如果该任务没有结束，则放回列表尾部
        except StopIteration:                    # 该任务结束，丢弃
            pass
```

```
from functools import partial


def task(id, n, m):                              # 模拟任务模板
    for i in range(n, m):
        print(f"{id}: {i}")
```

```
        yield                                   # 主动调度


t1 = partial(task, 1, 10, 13)
t2 = partial(task, 2, 30, 33)
sched(t1, t2)
```

输出：

```
1: 10
2: 30
1: 11
2: 31
1: 12
2: 32
```

5.4 函数式编程

函数式编程（Functional Programming）中的“函数”更趋近于数学概念，以计算表达式替代命令式语句。强调逐级结果推导，而非执行过程。

函数式编程要求函数为第一类型，可作为参数和返回值传递。需要时，可用闭包构成带有上下文状态的逻辑返回。其通常不使用独立变量，所有状态以参数传递，用嵌套或链式调用代替过程语句。最好是没有外部依赖的纯函数，且参数不可变，仅以结果带入下一级运算。

命令式

```
x = a + b
y = x * 2
```

函数式

```
mul(add(a, b), 2)

add(a, b).mul(2)
```

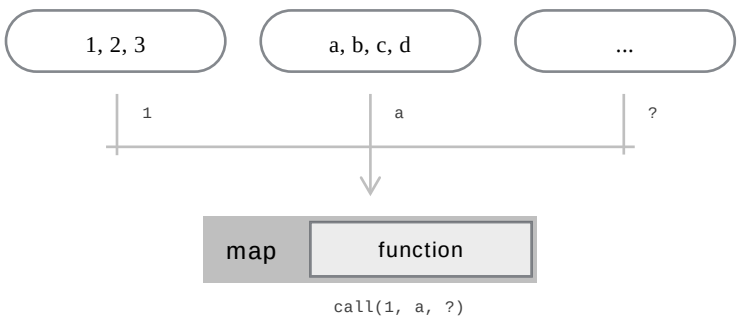
相比于命令式代码，函数式风格更精简，更易组合，且可替换。其对单元测试和并发编

程友好，引用透明且没有副作用。

> 函数式编程的三大特征：第一类型函数、不可变数据、尾递归优化。
>
> 函数式编程只是一种编程范式，多数语言都支持或部分支持它。

迭代

以迭代器方式，用指定函数处理数据源。相比于推导式，它能从多个数据源中平行接收参数，直至最短数据源迭代完成。




```
>>> x = map(lambda a, b: (a, b), [1, 2, 3], "abcd")

>>> list(x)                                          # 将迭代器对象转换为列表
[(1, 'a'), (2, 'b'), (3, 'c')]
```

> Python 3 以最短数据源为截止，而不像 Python 2 那样以 None 补位。
>
> 如果参数已经完成并行组合，可使用 itertools.starmap 代替。

聚合

平行从多个数据源接收参数，聚合成元组，直到最短数据源迭代结束。

```
>>> x = zip([1, 2, 3], "abcd", (1.1, 1.2))
```

```
>>> list(x)
[(1, 'a', 1.1), (2, 'b', 1.2)]
```

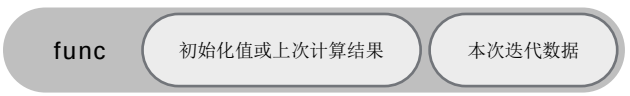
其常被用来构造字典。

```
>>> kv = zip("abcd", range(100, 200))

>>> dict(kv)
{'a': 100, 'b': 101, 'c': 102, 'd': 103}
```

> 如果以最长的数据源为准，则可使用 itertools.zip_longest。

累积

迭代数据源，将结果带入下次的计算，这适合完成统计或过滤操作。




```
>>> import functools

>>> def calc(ret, x):
        print(f"ret = {ret}, x = {x}")
        return ret + x
```

```
>>> functools.reduce(calc, [1, 2, 3])            # 没有初始化值，直接将第一数据当结果
ret = 1, x = 2
ret = 3, x = 3
6
```

```
>>> functools.reduce(calc, [1, 2, 3], 100)       # 需要对第一数据和初始化值进行计算
ret = 100, x = 1
ret = 101, x = 2
ret = 103, x = 3
106
```

如果没有初始值，则跳过第一数据，直接将其当作下次计算的参数。

同 Python 2 内置的 reduce 函数。

过滤

使用指定函数对数据进行迭代过滤。

函数返回的布尔值决定数据的去留。

参数为 None 时相当于 bool 函数，移除所有 False 元素。

```
>>> x = filter(lambda n: n % 2 == 0, range(10))

>>> list(x)
[0, 2, 4, 6, 8]
```

```
>>> list(filter(None, [0, 1, "", "a", [], (1,)]))
[1, 'a', (1,)]
```

判断

判断在一系列数据中，全部或某个元素为真值。

```
>>> all([1, "a", (1,2)])
True

>>> any([0, "", (1,2)])
True
```

迭代会短路。比如 all 遇到第一个 False，则立即终止。

```
>>> def a():
        print('a')
```

```
        return True

>>> def b():
        print('b')
        return False

>>> def c():
        print('c')
        return True
```

```
>>> all(map(lambda m: m(), (a, b, c)))
a
b
False

>>> any(map(lambda m: m(), (a, b, c)))
a
True
```

标准库 itertools 拥有大量针对迭代器的操作函数，推荐使用。

第 6 章 模块

6.1 定义

模块（module）是顶层代码组织单元，其提供大粒度封装和复用。

通常每个模块对应一个源码文件。从某些角度来看，模块就像更大规模的类。其中定义的变量、函数、类型等，都属于其私有成员。

模块也可用 C 或 Cython 编写，以绕开解释器和 GIL 限制，获得更好的执行性能与并发支持。

模块在首次导入（import）时，被编译成字节码。随后解释器开始创建模块实例，执行初始化语句，构建内部成员。模块不仅是代码组织形式，还是运行期对象，其为成员提供全局名字空间。

无论被导入多少次，每个模块在整个解释器进程内都仅有一个实例存在。随后，其也不会监测源文件是否变动。重复导入只是引用已存在的实例，并不会再次执行初始化过程。

demo.py

```
x = 1234
```

```
def hello(): pass
class User: pass
```

```
>>> import demo

>>> isinstance(demo, types.ModuleType)
True
```

当然，其也能像普通类型那样直接创建模块实例，未必非得有源码文件。

```
>>> types.ModuleType("abc")
<module 'abc'>
```

初始化

所谓初始化过程，就是将模块里的代码按序执行一遍。正好，已编译的字节码缓存文件可作为反汇编目标。

```
>>> import io, marshal, dis
>>> import demo

>>> f = open(demo.__cached__, "rb")        # 以二进制方式打开缓存文件
>>> f.seek(12, io.SEEK_SET)                # 跳过 magic 等头部信息
>>> code = marshal.load(f)                 # 反序列化，还原代码对象
```

```
>>> dis.dis(code)
  3           0 LOAD_CONST               0 (1234)
              2 STORE_NAME               0 (x)

  6           4 LOAD_CONST               1 (<code object hello>)
              6 LOAD_CONST               2 ('hello')
              8 MAKE_FUNCTION            0
             10 STORE_NAME               1 (hello)

  9          12 LOAD_BUILD_CLASS
             14 LOAD_CONST               3 (<code object User>)
             16 LOAD_CONST               4 ('User')
             18 MAKE_FUNCTION            0
             20 LOAD_CONST               4 ('User')
```

```
              22 CALL_FUNCTION            2
              24 STORE_NAME               2 (User)
              26 LOAD_CONST               5 (None)
              28 RETURN_VALUE
```

初始化过程很简单：普通语句直接执行，类似于 def、class 等则创建函数和类型对象。最终这些成员都被保存到模块的全局名字空间内。

```
>>> vars(demo)
{
    '__cached__': '__pycache__/demo.cpython-36.pyc',
    '__file__': 'demo.py',
    '__name__': 'demo',
    'User': demo.User,
    'hello': <function demo.hello>,
    'x': 1234, ...
 }
```

> 以示例中的全局变量 x 为例，初始化过程为其赋予一个初始值。而后的执行过程中，它可能会发生变更。如果每次导入都重新执行初始化过程，那么势必导致该变更状态丢失。所以说，重复导入只是引用，不会再次执行。

名字空间

模块的全局名字空间对应__dict__属性，其在内部不能被直接访问，且不会被 dir 输出。

```
>>> vars(demo) is demo.__dict__
True
```

> 函数 vars 返回目标__dict__属性。参数为空时，类似于 locals 返回当前名字空间。
>
> 而 dir 则返回目标，或当前名字空间可访问的名字列表。

当 def 创建函数对象时，会将所在模块的名字空间作为构造参数。这意味着，无论后续将该函数传递到何处，还是“绑定”给其他模块，均不能改变函数内部 globals 总是返回出生地的名字空间。

demo.py

```
def test():
    return globals()
```

```
>>> demo.test() is vars(demo)              # 在任何地方调用，函数依然返回出生地的名字空间
True
```

```
>>> m = types.ModuleType("abc")
>>> m.test = demo.test                     # 即便“绑定”给其他模块，依然无法改变这一点

>>> m.test() is vars(demo)
True

>>> m.test.__module__
'demo'
```

名字

可通过__name__、__file__获知所在模块的信息。或者，利用__module__属性返回类型、函数等对象的定义模块名称。

```
>>> x = demo.hello
>>> x.__module__
'demo'
```

> 建议使用 inspect.getmodule 获取目标对象定义模块引用。

在正常情况下，模块名字对应源文件名（不含扩展名）。这里仅有一个例外，就是当模块作为程序入口时，会被赋予“__main__”名字。

demo.py

```
print(__name__)
```

```
$ python demo.py                    # 入口文件
__main__

$ python -c "import demo"           # 普通导入
```

```
demo
```

如此，我们只要编写了如下语句，就可让其自动适应身份变化。

```
def main():
    test()

if __name__ == "__main__":
    main()
```

> 创建语句 def 并不会执行 main 函数，只有作为入口时才会进入流程。如果作为普通模块导入，则因名字不匹配而被忽略。另外，不建议将 main 函数内容直接写到 if 语句块内，因为这会直接操作全局名字空间，可能引发不必要的错误。

6.2 导入

在多数语言中，多源码文件共享全局环境。但在 Python 里不同，每个模块文件都有自己独立的全局名字空间，这是其内部代码访问边界。

依照 LEGB 规则，除包含内置函数的__builtins__模块外，所有名字搜索都不能超出当前模块。所以任何外部目标，都必须提前导入当前名字空间，否则无法访问。

完整导入步骤：

1．搜索目标模块文件。

2．按需编译目标模块。

3．创建模块实例，执行初始化。

4．将模块实例保存到全局列表。

5．在当前名字空间建立引用。

模块实例一旦创建，就保存到 sys.modules，后续导入直接引用。

内置模块__builtins__由解释器自动导入。

```
>>> for i in range(3):
        import sys                                  # 重复导入，不会新建实例
        print(id(sys))

4480974584
4480974584
4480974584
```

```
>>> dis.dis(compile("import sys", "", "exec"))
        4 IMPORT_NAME              0 (sys)          # 导入模块
        6 STORE_NAME               0 (sys)          # 在当前名字空间建立关联
```

6.2.1 搜索

导入所使用的模块名不包含路径信息，这需要系统提供搜索方式和匹配规则。其中搜索路径，由解释器在启动时，按优先级整理到 sys.path 列表中。

搜索路径列表：

1．程序根目录。

2．环境变量（PYTHONPATH）设定的路径列表。

3．标准库目录。

4．第三方扩展库等附加路径（site-specific）。

附加路径由 site 模块添加，在解释器启动时自动执行（可用参数-S 禁用）。

除系统 site-packages 外，还包括用户相关目录。

```
$ python3 -c "import sys, pprint; pprint.pprint(sys.path)"
['',                                                    # 应用程序根目录
 '/.../lib/python36.zip',                               # 标准库
 '/.../lib/python3.6',
 '/.../lib/python3.6/lib-dynload',
 '/.../lib/python3.6/site-packages']                    # 系统扩展库目录
```

```
$ python3 -S -c "..."                                   # 禁用 site，缺少 site-packages
['',
 '/.../lib/python36.zip',
 '/.../lib/python3.6',
 '/.../lib/python3.6/lib-dynload']
```

```
$ PYTHONPATH=/usr/local/a:/usr/local/b python3 -c "..."  # 环境变量，注意优先级
['',
 '/usr/local/a',
 '/usr/local/b',
 '/.../lib/python36.zip',
 '/.../lib/python3.6',
 '/.../lib/python3.6/lib-dynload',
 '/.../lib/python3.6/site-packages']
```

虽然可在运行期向 sys.path 添加搜索路径，但对扩展库而言，更简便的方式是路径配置文件。将所有要添加的路径按行保存在.pth 文本文件内，并放到 site-packages 等目录中，剩余的事情由 site 完成。

site-packages/demo.pth

```
# comment                                               # 注释

/usr/local/go                                           # 绝对路径
mypkg                                                   # 相对路径
```

```
$ python -m site                                        # 查看完整搜索路径列表

sys.path = [
    '/Users/qyuhen/test',                               # 应用程序根目录
    '/.../lib/python36.zip',                            # 标准库
    '/.../lib/python3.6',
```

```
    '/.../lib/python3.6/lib-dynload',
    '/.../lib/python3.6/site-packages',             # 系统扩展库目录
    '/usr/local/go',                                # 路径配置文件添加目录
    '/.../lib/python3.6/site-packages/mypkg',
]

USER_BASE: '/Users/qyuhen/.local' (doesn't exist)
USER_SITE: '/Users/qyuhen/.local/lib/python3.6/site-packages' (doesn't exist)
ENABLE_USER_SITE: False
```

> 提示：site 会过滤掉配置文件中重复，以及不存在的路径。

在整理好搜索路径列表后，接下来需要进一步匹配模块文件名。但问题是，语言规范并未定义不同扩展名的同名模块匹配次序，这也就无法保证所有解释器的选择结果一致。为安全起见，不要在同一路径下放置同名模块。

- 源码或字节码文件（.py、.pyc）。
- 包目录名。
- 其他语言编写的扩展模块（.dll、.so 等）。

6.2.2 编译

确定模块文件路径后，解释器优先选择已编译过的字节码文件，这有助于提升载入性能。当然，执行之前，须读取头信息，确认源文件是否更新，以便重新编译。

> 从 Python 3.2 开始，字节码缓存文件被整体转移到__pycache__子目录。相比于以前的版本，这避免了对源码目录的污染，有利于管理和维护。
>
> 缓存文件名包含解释器、版本和优化信息，且不再使用.pyo 扩展名。如此，不同版本的解释器缓存文件可和谐共处。
>
> 我们可使用-B 参数阻止生成字节码缓存文件。

```
$ python -c "import demo"
```

```
$ ls __pycache__
demo.cpython-36.pyc
```

```
$ python -OO -c "import demo"

$ ls __pycache__
demo.cpython-36.opt-2.pyc
```

无源码部署

单就运行而言，源码文件不是必需的。这是因为解释器需要的字节码及相关元数据，要么编译后存储在内存，要么就保存在字节码文件中。

可移除源码，改为直接部署字节码文件（.pyc），这算是一种轻度保护。

> 不能直接使用__pycache__下的文件，它的文件命名方式不适合作为直接部署使用。
>
> 缓存文件很容易被 dis 反汇编，并不能有效地保护代码安全。

main.py

```
import demo
demo.hello()
```

demo.py

```
def hello():
    print("hello, world!")
```

这里直接使用 compileall 编译所有源文件。

```
$ python -m compileall -b -l *.py        # 编译所有文件（以原文件名保存，不包括自目录）
Compiling 'demo.py'...
Compiling 'main.py'...

$ mv *.pyc release/                      # 将生成的.pyc 文件转移到独立目录
$ cd release/
$ ls
demo.pyc main.pyc
```

```
$ python main.pyc                          # 使用解释器执行
hello, world!
```

因为.pyc 是二进制文件，就算 main.py 头部有 Shebang 也没用，也只能以解释器参数方式执行。

虽然字节码是平台中立格式，但.pyc 头部的 magic number 却保存了解释器版本，所以要使用相同版本（大版本号）的解释器执行。

6.2.3 引用

可同时导入多个模块，或模块中的多个成员。当名字发生冲突时，可用 as 命名别名。

```
>>> import sys, inspect as reflect
>>> from string import ascii_letters as letters, ascii_lowercase as lowercase
```

```
>>> %whos                                  # IPython 命令，查看当前环境变量信息

Variable    Type      Data/Info
------------------------------
letters     str       abcde...
lowercase   str       abcde...
reflect     module    <module 'inspect'>
sys         module    <module 'sys'>
```

别名仅在当前名字空间有效，其并不影响被导入的模块或其成员。别名还可用来做缩写处理。

如果导入的成员过多，可用括号或续行符分成多行。

在多数编码规范中，通常建议每行仅导入一个模块，且不在全局名字空间导入模块成员。

导入语句放在文件头部，以“标准库、扩展库，当前程序模块”分块排列。

```
import sys                 # 标准库
import inspect
```

```
import psutil              # 第三方扩展库
import tornado

import db                  # 应用程序模块
import logic
```

至于成员导入，其名字和其他模块，或当前模块成员冲突的可能性较大。且对模块重新载入有影响（见下面的内容），故不建议在模块级别使用。至于以星号导入目标模块“全部”成员的行为，更是被规范所禁止。

对成员而言，合适的名字通常就是一些常用单词。如果是以模块为前缀的全名，则通常可避免冲突。但若直接以成员导入，则其可能被“同名覆盖”。

建议参考 *Google Python Style Guide* 等文档，养成规范而统一的编码风格。另外，还可使用 isort 工具自动整理导入语句。

```
from sys import version
print(version)

version = "ver: 1.0"              # 在当前名字空间内被覆盖，同一名字仅有一个关联
print(version)
```

在函数内部，因其名字空间范围小，且生命周期较短，故影响也小。如此，使用成员导入可减少编码，且还能在一定程度上提升性能。

Python 3 不允许在函数内使用星号导入，只能用于模块级别。

```
def test():
    from sys import version
    print(version)
```

下面对比模块导入和成员导入指令的差异性。

```
def a():
    import sys
```

```
    print(sys.version)

def b():
    from sys import version
    print(version)
```

```
>>> dis.dis(a)
  3           8 LOAD_GLOBAL              1 (print)
             10 LOAD_FAST                0 (sys)             # 需要两条指令引用 version
             12 LOAD_ATTR                2 (version)
             14 CALL_FUNCTION            1

>>> dis.dis(b)
  7          12 LOAD_GLOBAL              2 (print)
             14 LOAD_FAST                0 (version)         # 单条指令，且是 FAST
             16 CALL_FUNCTION            1
```

如果是多次调用，那么对比效果就比较明显。

```
def a():
    import sys
    for i in range(100):
        s = sys.version
    return s

def b():
    from sys import version
    for i in range(100):
        s = version
    return s
```

```
>>> %timeit a()
6.02 µs ± 253 ns per loop

>>> %timeit b()
4.18 µs ± 261 ns per loop
```

导出列表

星号导入是将目标模块名字空间内的所有成员全部引入，这里面必然有很多用不到的东

西。但作为模块开发者，我们无法阻止用户使用星号。结果就是调用方的名字空间被污染，这显然不是我们愿意看到的。另外，我们编写的模块也会导入其他模块，它们会被裹挟着一同进入调用方的名字空间。

demo.py

```
import sys

x = 1234
def hello(): pass
```

main.py

```
from demo import *
print(dir())
```

```
$ python main.py              # demo 的成员，以及它导入的 sys，都进入调用者 main 的名字空间
['hello', 'sys', 'x', ...]
```

简单做法是模块成员私有化，也就是为名字添加下画线前缀。私有成员不会被星号导入，但我们导入的其他模块呢？私有别名又如何？

demo.py

```
import sys as _sys          # 私有别名?

_x = 1234                   # 私有成员
def hello(): pass
```

```
$ python main.py
['hello', ...]
```

显然这种写法既不方便也不美观。正确做法是在模块内添加__all__声明，明确指定那些可以被星号导入的成员名字列表。为空时，表示不会导入任何成员。

demo.py

```
__all__ = ["hello"]
import sys
x = 1234
def hello(): pass
```

```
$ python main.py                # 仅__all__指定成员被星号导入
['hello', ...]
```

当然，无论是私有成员，还是__all__导出列表，都不影响显式成员导入。

demo.py

```
__all__ = []

import sys

_x = 1234
def hello(): pass
```

main.py

```
from demo import _x, hello
print(dir())
```

```
$ python main.py
['_x', 'hello', ...]
```

> Python 的很多“私有”规则只是调用约定，并不是真正意义上的权限设置。

动态导入

使用 import 的前提是明确知道目标模块的名字。但某些时候，只有在运行期才能动态获知，这就需要以其他方式导入。

方法一：使用 exec 动态执行，随后从 sys.modules 中获取模块实例。

```
import sys

def test(name):
    exec(f"import {name}")
    m = sys.modules[name]
    print(m)
```

```
test("inspect")
```

方法二：使用 importlib 库。

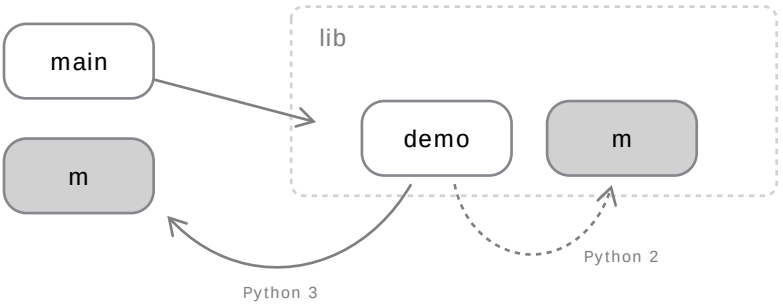
```
import importlib

def test(name):
    m = importlib.import_module(name)
    print(m)
```

两种方法都会将模块实例保存到 sys.modules 内。

笔者不建议使用__import__函数。作为 import 语句的内在实现，其行为在 Python 2 和 Python 3 下存在差异，这可能导致意外错误。

```
/
|
+--- main.py                      # import lib.demo
|
+--- m.py                         # print(__file__)
|
+--- lib/
      |
      +--- __init__.py            # 空文件
      |
      +--- demo.py                # import m
      |
      +--- m.py                   # print(__file__)
```

```
$ python2 main.py
/lib/m.py

$ python3 main.py
/m.py
```



其原因是两个版本的搜索起始路径不同。Python 2 从当前目录开始，那么 demo 优先选择的自然是同一目录下的 m，而 Python 3 总是从应用程序根目录开始。

我们可使用 import_module 进行代替，它在两个版本中的行为一致，都从根目录开始。

某些时候，模块需要同时支持 Python 2 和 Python 3，这时可用异常保护处理导入语句的差异。

```
try:
    from urlparse import urljoin
    from urllib2 import urlopen
except ImportError:
    # Python 3
    from urllib.parse import urljoin
    from urllib.request import urlopen
```

重新载入

模块被导入后，解释器就不再关心源文件是否被修改过，这对简单或者已趋于稳定的程序没什么影响。但对于那些需要在线测试，或频繁变更业务逻辑的互联网应用，我们就不得不设法在不重启进程的情况下，对某些模块实现热更新。

至于如何监测源文件的变化、如何启动热更新，这些内容均不在本书的讨论之列。这里，我们尝试不同模块的重载方案，探知其实现机制和存在的问题。

既然模块实例保存在 sys.modules 列表里，那么是否可通过移除实例引用，使其被回收

后再重新冷导入呢？

```
demo.py
    x = 1234
```

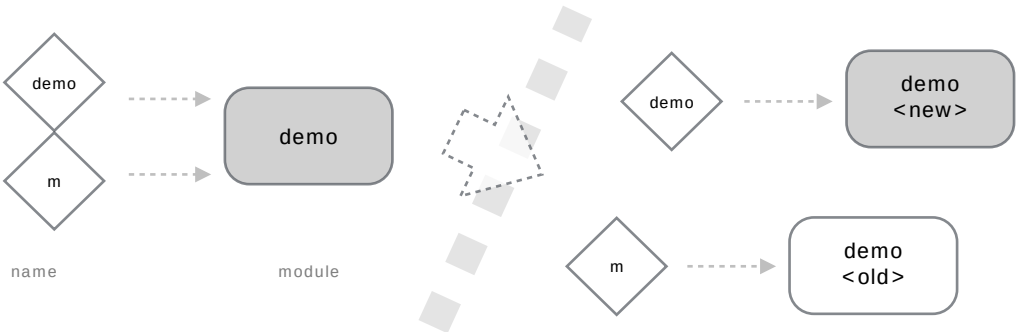
```
IPython
    >>> import demo

    >>> demo.x
    1234

    >>> del sys.modules["demo"]            # 从 sys.modules 移除，使其回收
    >>> !echo "x = 999" > demo.py          # 修改 demo.py

    >>> import demo                        # 重新导入
    >>> demo.x                             # 更改生效
    999
```

从结果来看，似乎没有什么问题。可如果有其他引用存在呢？

```
IPython
    >>> import demo

    >>> m = demo                           # 另一引用
    >>> m.x
    999

    >>> del sys.modules["demo"]            # 移除
    >>> !echo "x = 888" > demo.py          # 修改源文件

    >>> import demo                        # 重新导入
    >>> demo.x
    888

    >>> m.x                                # 对另一引用 m 没有影响，热更新失败
    999
```

其失败的原因很简单。因为 m 的存在，导致旧 demo 模块实例不会被回收。如此一来，更新后它们各自引用不同的模块实例（即便旧模块已不在 sys.modules 列表里）。

```
>>> m is demo
False
```



标准库为我们提供了另外一种方案 importlib.reload。它直接原址（memory in-place）替换模块内容，这样就可让所有引用都指向新的模块实例。

```
>>> import demo

>>> m = demo                          # 另一引用
>>> m.x
888

>>> !echo "x = 666" > demo.py         # 修改源文件
>>> importlib.reload(demo)            # 重载模块

>>> demo.x                            # 热更新成功
666

>>> m.x                               # 其他引用也同步更新
666

>>> demo is m
True
```

这种方案不变动内存地址，直接替换内容，这就不会影响原有引用。但问题是，这种方法只对模块引用有效，但对成员引用无法更新，因为 reload 不会递归修改成员。

```
>>> import demo

>>> x = demo.x                        # 引用另一模块的成员
```

```
>>> x
666

>>> !echo "x = 777" > demo.py          # 修改源文件
>>> importlib.reload(demo)             # 重载模块

>>> demo.x                             # 热更新成功
777

>>> x                                  # 但对原成员引用无效
666

>>> demo.x is x
False
```

有鉴于此，应避免直接引用其他模块成员，而应总是通过模块间接访问。当然，在函数内部，成员引用的生命周期较短，对热更新影响有限。

另外，模块重载会重新初始化，这会导致状态丢失，并可能引发连带错误，我们须慎重对待。

6.3 包

如果说模块用于组织代码，那么包就用来组织模块。

将多个源文件放置于同一目录，就构成了包。包能隐藏内部文件的组织结构，而仅暴露必要的用户接口，毕竟不是所有模块都对外提供服务。

与模块类似，包名直接对应目录名。包名可以是产品名称，以及公司或组织结构的名称等。

包为模块提供更详细的名字前缀。相比于模块名 db，有包前缀的全名 orm.db 更不易冲突。

虽然形式上只是一个目录，但包也是运行期对象，它有自己的名字空间。

```
/
|
+--- main.py
|
+--- lib/                        (package)
      |
      +--- demo.py               (package.module)
```

```
>>> import lib

>>> lib
<module 'lib' (namespace)>

>>> vars(lib)
{
    '__name__': 'lib',
    '__package__': 'lib',
    '__path__': _NamespacePath(['test/lib']), ...
}
```

从包的名字空间可以看出，仅导入包并不能直接访问其内部模块，须显式导入。

```
>>> import lib.demo

>>> lib.demo.__package__
'lib'

>>> lib.demo.__name__
'lib.demo'
```

6.3.1 初始化

包是另类的模块实例，其对应源文件__init__.py。虽然该文件在 Python 3 里不再是必需的，但其可用来执行某些初始化操作。比如提供对外接口，解除用户对内部模块的直接依赖。

```
lib/
 |
 +--- __init__.py
 |
 +--- demo.py
```

```
>>> import lib

>>> lib.__file__
'lib/__init__.py'
```

初始化文件在包或其内部模块首次导入时自动执行，且仅执行一次。

__init__.py
```
print("init")
```

demo.py
```
print("demo")
```

```
>>> import lib.demo
init
demo

>>> import lib
```

> 重载（reload）包内模块，不会再次执行初始化文件，但重载包会。

还可在包内创建__main__.py 文件，作为直接执行时的入口。

__main__.py
```
print("main")
```

```
$ python -m lib                 # 以包方式运行，自动执行初始化文件
init                            # 先导入包，后执行__main__.py
main

$ python lib                    # 以普通程序方式运行，对应__main__入口模块
main                            # 不会自动执行初始化文件，除非显式导入__init__
```

```
$ python -c "import lib"        # __main__.py 对普通导入没有影响
init
```

隐藏结构

既然初始化文件构成了包名字空间，那么只要将要公开给用户的模块或成员导入进来，就可以解除用户对具体模块的依赖。

```
demo.py
    def hello():
        print("hello, world!")
```

```
__init__.py
    from .demo import hello
```

```
    >>> import lib

    >>> lib.hello()
    hello, world!
```

如此一来，用户只依赖包和初始化文件中导入的特定成员。无论我们如何重构代码和重新组织模块文件，都不影响用户调用接口。即便将公开成员升级到新版本，也可用别名方式映射到原名字，实现多版本的单一名称。

```
demo.py
    def hello():
        print("hello, world!")

    def hello2():
        from sys import version
        print(version)
        hello()
```

```
__init__.py
    from .demo import hello2 as hello
```

还可更进一步，将整个包压缩成 ZIP 文件分发。

```
lib.v1.zip                          任意文件名，可添加版本等信息
 |
 +--- lib/                          必须包含包目录
       |
       +--- __init__.py
       |
       +--- demo.py
```

> 如果只打包模块，则可直接将模块文件放到压缩包的根目录下。

```
$ zip -r -o lib.v1.zip lib                  # 压缩后，可将原目录重命名，以免影响下面的测试
  adding: lib/ (stored 0%)
  adding: lib/__init__.py (deflated 9%)
  adding: lib/demo.py (deflated 37%)
```

使用前，须将.zip 文件的路径添加到 sys.path，这可动态添加或使用路径配置文件。

```
>>> sys.path.append("./lib.v1.zip")

>>> import lib

>>> lib.__file__                            # 指向压缩包内的文件
'./lib.v1.zip/lib/__init__.py'
```

> 因解释器不会向 .zip 中写入缓存文件，故建议连带 .pyc 一起打包，以加快导入速度。

```
$ python -m compileall -b lib/*.py          # 不能使用__pycache__
$ zip -r -o lib.v1.zip lib/*.pyc            # 仅打包.pyc 文件，注意包含包目录
```

也可将应用程序进行打包，不过须在根目录下放置 __main__.py 作为执行入口。

```
app.zip
 |
 +--- __main__.py
 |
 +--- lib/
       |
       +--- __init__.py
       |
       +--- demo.py
```

__main__.py

```
print("main")

import lib
lib.hello()
```

```
$ zip -r -o app.zip __main__.py lib

$ python app.zip
main
hello, world!
```

导出列表

同样可在初始化文件中添加__all__星号导出成员列表。除当前文件成员外，还可指定要导出的模块名字。

> 不用 import 语句显式导入，只需在列表中添加要导出的模块名字即可。
>
> 无论是隐藏结构（包括 .zip），还是__all__导出列表，都不影响用户显式导入包内模块。

__init__.py

```
__all__ = ["x", "demo"]            # 可被星号导出的模块和本地成员

x = 100
y = 200
```

```
>>> from lib import *

>>> dir()
["demo", "x", ...]
```

6.3.2 相对导入

通常将包放置于应用程序的根目录，或其他系统目录。但是，搜索路径列表中并不包括包目录自身，这就导致在包内访问同级模块时，发生找不到文件的状况。

__init__.py

```
import demo
demo.hello()
```

demo.py

```
def hello():
    print("hello, world!")
```

```
>>> import lib

lib/__init__.py in <module>
      1
----> 2 import demo
      3 demo.hello()

ModuleNotFoundError: No module named 'demo'
```

就算在包内导入，import 语句也严格按照搜索路径列表查找。它会搜索应用程序的根目录，但不检查包目录。这就是引发错误的原因，除非全名导入。

Python 2 优先搜索当前路径，这也就是 relative-then-absolute 顺序。

Python 3 移除了该设定，其严格按 sys.path 进行搜索。它能从应用根目录找到 lib，但找不到 demo。

__init__.py

```
import lib.demo
lib.demo.hello()
```

在包内使用全名的绝对路径自然不够友好。为此，Python 3 引入了相对路径的概念，以点前缀表达当前或上级包。

```
/
 |
 +--- lib/
       |
       +--- __init__.py
       |
       +--- demo.py
       |
       +--- sub/
             |
             +--- test.py
```

lib/sub/test.py

```
from ..demo import hello              # 从上级包的 demo 模块导入 hello
```

lib/__init__.py

```
from . import demo                    # 从当前包导入 demo 模块
from .demo import hello               # 从当前包的 demo 模块导入 hello
from .sub import test                 # 从当前包的 sub 子包导入 test 模块
```

> 相对导入只能用于 from 子句，其可导入模块或成员。
>
> 单个 . 表示当前包，... 表示上级包，... 表示更上一级的包，依此类推。

相对路径解除了对包名的依赖，其还可用来解决名字冲突。

> 比如导入 string 时，如果使用相对导入则明确表示选择包内（或上级）模块，否则就是选择标准库的同名模块。
>
> 相比于 Python 2 优先选择当前路径的隐式设定，Python 3 的显式做法要更清晰。其避免了在 Python 2 下导入标准库 string，却反而优先选择本地同名模块的尴尬。显式规则总要好过隐式设定。

优先级

同名除让解释器困扰外，也给代码维护带来了潜在的风险。当相同名字的包和模块出现

在同一路径下，那么解释器将按如下顺序匹配。

1．如果包有初始化文件（包括空文件），则导入包。

2．没有初始化文件，则.py 或.pyc 模块优先。

如果没有同名冲突，那么还是按搜索路径规则走。所以说，应尽可能避免同名问题。

```
(1)                              (2)

/                                /
|                                |
+--- lib.py                      +--- lib.py ★
|                                |
+--- lib/                        +--- lib/
      |                                |
      +--- __init__.py ★                +--- demo.py
      |
      +--- demo.py
```

6.3.3 拆分

当包内的模块文件过多时，可建立子包分组维护，但这需要修改内部的相对导入路径。还有一种方法，同样是将文件分散到多个子目录下，但它们依然属于同一级别的包成员。

先将原本放在同一目录下的模块文件分别转移到各自的分组子目录中。

为让它们继续以包成员的形式存在，须在包__init__.py 中修改__path__属性。

包__path__的作用类似于 sys.path，实现包内搜索路径列表，并使用相同匹配规则。在该搜索列表中默认为包的全路径，只需将子目录的全路径添加进去即可。因子目录中的各模块依然属于原包级别，故模块间的相对导入无须更改。

```
/                               /
 |                               |
 +--- lib/                       +--- lib/
       |                               |
       +--- __init__.py                +--- __init__.py
       |                               |
       +--- demo.py                    +--- a/
       |                               |    |
       +--- hello.py                   |    +--- demo.py
                                       |
                                       +--- b/
                                            |
                                            +--- hello.py
```

__init__.py

```
import os.path

__path__.extend(os.path.join(__path__[0], d) for d in ("a", "b"))

from . import demo                          # 包内的各模块依然被视作同一级别
from . import hello

print(__path__)
print(demo.__file__)
print(hello.__file__)
```

```
>>> import lib
['/Users/qyuhen/test/lib',                  # __path__
 '/Users/qyuhen/test/lib/a',
 '/Users/qyuhen/test/lib/b']

/Users/qyuhen/test/lib/a/demo.py
/Users/qyuhen/test/lib/b/hello.py
```

因为__path__添加的是全路径，所以这意味着可将子目录放到任意位置，甚至可将其他包的内容引入进来。但这可能导致混乱，故此不推荐使用。

另外，对包内文件的迭代可使用标准库 pkgutil 的相关函数。

第 7 章 类

7.1 定义

类（class）封装一组相关数据，使之成为一个整体，并使用方法持续展示和维护。这有点像把零件组装成整车提供给用户。无须了解汽车的内部结构和工作原理，只要知道方向盘、刹车和油门这些外在接口就可以正常行驶。

与函数类似，类也是一种小粒度复用单位，但其行为特征更为复杂。函数具有单一入口和出口，可以完成一次计算过程。类从构造开始就面临多个方法。这些方法在不同调用次序下，会产生不同的结果。这就好像同一辆车，因为驾驶员的不同驾驶习惯就会有不同的运动轨迹。而函数更像自动烤面包机，放入和拿回不会因人而有太大差异。

如果说函数属于数学范畴，那么类就属于生物学范畴。类擅长对有持续状态、有生命周期、有遗传特征的物体进行抽象模拟。无论模拟的是动物还是植物，在保留种族共性的前提下，个体总有属于自己的体貌特征和行为差异性。如此一来，函数就像机械加工，着重于处理过程；类则关注于数据本身，使其“活”过来。

类存在两种关系：继承（inhertiance，is-a）自某个族类，组合（composition，has-a）了哪些部件。前者可用来表达本车属于某厂的哪个车族系列，除继承原车系的技术和优势

外，还基于哪些环境进行了改进。而后者则表述该车使用了哪些零部件，比如最新的发动机。

作为一种复合结构，类与模块有相似之处。但不同之处在于：

- 类可生成多个实例。
- 类可被继承和扩展。
- 类实例的生命周期可控。
- 类支持运算符，可按需重载。

这些是模块没有或不需要的。同时，模块的粒度大，其可用来提供游戏场景级别的解决方案，而类则是该场景下的特定家族和演员。

类可以有多个外在方法，但其应专注于单个目标。即便用户类需要加密手段保存信用卡等敏感数据，也不应有加密相关代码，而应选择组合安全数据类型，以便替换和分离测试。

在某些语言中，类的用途被放大，并形成了一种名字空间的概念。比如将多个函数附加到一个静态类上，形成函数套装。这与本书所提及的类有所不同，此处特指自定义复合数据类型，这是一个缩小的概念。至于静态类这种用途，完全可用模块替代。

建议将类与业务逻辑分离。逻辑代表业务处理过程，其通常被设计成无状态，用上下文传递数据。业务逻辑讲究效率和并发性，以及相对固定的执行过程。从这一点看，逻辑适合以函数分段实现，按特定顺序组装，并以模块存储。这些特点都不适合以数据为核心，且注重家族遗传，用多实例体现个性的类来实现。毕竟，逻辑以组装和替换实现持续改良和升级，有稳固测试流程，传承并非重点，也没有多实例需求。

创建

定义类，以此为工厂制造个体实例。

```
class User:
    pass
```

可在函数内定义，以限制其作用范围。

关键字 class 同样是运行期指令，用以完成类型对象的创建。我们准备了相对“完整”的示例，下面查看具体的创建过程。

```
def test():
    class X:
        data = 100
        def get(self): return self.data
```

```
>>> dis.dis(test)
  2           0 LOAD_BUILD_CLASS
              2 LOAD_CONST               1 (<code X>)       # test.__code__.co_consts[1]
              4 LOAD_CONST               2 ('X')
              6 MAKE_FUNCTION            0                  # 创建 X 函数
              8 LOAD_CONST               2 ('X')
             10 CALL_FUNCTION            2                  # 调用__build_class__函数
             12 STORE_FAST               0 (X)
```

从反汇编结果看，先创建 X 函数，其内容是属性设置和方法创建。随后，该函数被当作参数传递给 builtins.__build_class__调用，这里其实就是元类（metaclass）执行所在。

在__build_class__调用 X 时，会提供一个字典作为其堆栈帧的名字空间，用于保存类型成员。完成后，将该字典连同名字和基类一起传递给元类，最终生成目标类型对象。

提示：元类用于创建类型对象。详细情形，请参考后续章节。

```
>>> dis.dis(test.__code__.co_consts[1])
  2           0 LOAD_NAME                0 (__name__)
              2 STORE_NAME               1 (__module__)
              4 LOAD_CONST               0 ('test.<locals>.X')
              6 STORE_NAME               2 (__qualname__)

  3           8 LOAD_CONST               1 (100)
             10 STORE_NAME               3 (data)

  4          12 LOAD_CONST               2 (<code object test>)
             14 LOAD_CONST               3 ('test.<locals>.X.get')
```

```
              16 MAKE_FUNCTION            0
              18 STORE_NAME               4 (get)
```

类型与实例

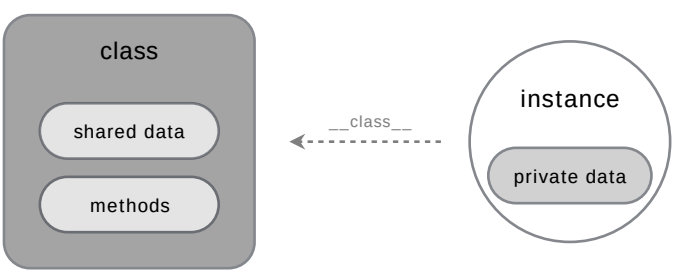
如果类型在模块中定义，那么其生命周期与模块等同。如果其被放在函数内，那么每次都是新建。即便名字和内容相同，也属于不同类型。

```
def test():
    class X: pass
    return X()
```

```
>>> a, b = test(), test()

>>> a.__class__ is b.__class__
False
```

> 函数内定义的类型对象，在所有实例死亡后，会被垃圾回收。

类型对象除用来创建实例外，也为所有实例定义了基本操作接口，其负责管理整个家族的可共享数据和行为模板。而实例只保存私有特征，其以内部引用从所属类型或其他祖先类查找所需的方法，用来驱动和展现个体面貌。



类型所创建的多个实例之间，除家族归属外，并无直接关系。无论从诞生到消亡，还是私有数据变更，都不影响类型或其他兄弟姐妹。

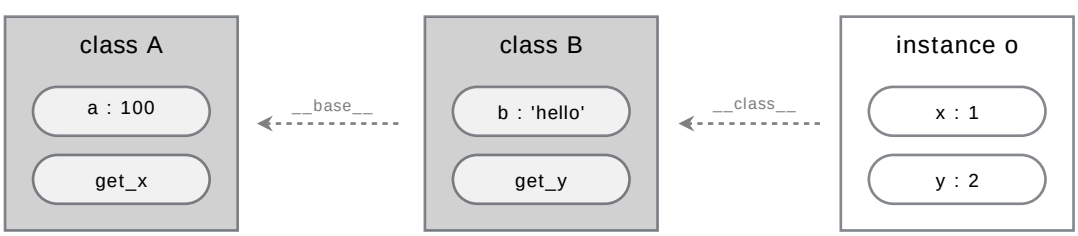
名字空间

类型有自己的名字空间，存储当前类型定义的字段和方法。这其中并不包括所继承的祖先成员。其同样以引用关联祖先类型，无须复制到本地。

```
class A:
    a = 100                             # 类字段

    def __init__(self, x):              # 实例初始化方法
        self.x = x                      # 实例字段

    def get_x(self):                    # 实例方法
        return self.x
```

```
class B(A):                             # 继承自 A
    b = "hello"

    def __init__(self, x, y):
        super().__init__(x)             # 调用父类的初始化方法
        self.y = y

    def get_y(self):
        return self.y
```

```
>>> o = B(1, 2)
```



实例会存储所有继承层次的实例字段，因为这些都属于其私有数据。方法算是函数变体，其本身并无状态，所以为大家所共享。但即便是从远古祖先那里继承的眉眼肤色，到了本人这里也是私有特征。如果也放到类型里共享，岂不千人一面？也正因为这些私有数据驱动，才让方法展现出不同的结果来。

```
>>> A.__dict__
mappingproxy({'__init__': <function A.__init__>,
              'a': 100,
              'get_x': <function A.get_x>, ...})
```

```
>>> B.__dict__
mappingproxy({'__init__': <function B.__init__>,
              'b': 'hello',
              'get_y': <function B.get_y>, ...})
```

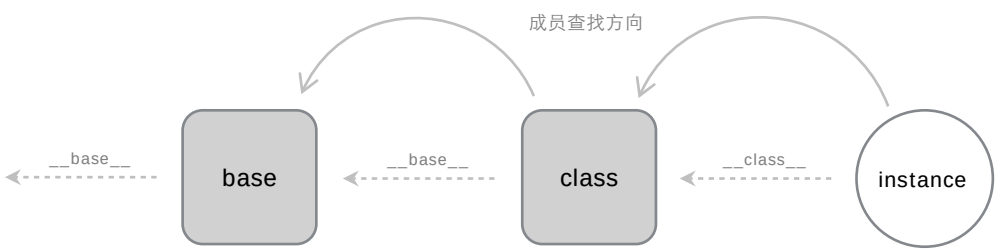
类型的名字空间返回 mappingproxy 只读视图，不允许直接修改。

```
>>> o.__dict__
{'x': 1, 'y': 2}

>>> dir(o)
['a', 'b', 'get_x', 'get_y', 'x', 'y', ...]
```

实例的名字空间是普通字典，可直接修改。

函数 dir 搜索所有可访问成员的名字，vars 直接返回__dict__属性。

当通过实例或类型访问某个成员时，会从当前对象开始，依次由近到远向祖先类查找。如此做的好处，就是祖先类的新增功能可直接“广播”给所有后代。



在继承层次的不同名字空间中允许有同名成员，并按顺序优先命中。

成员查找规则和 LEGB 不同。前者基于继承体系，后者基于执行作用域。

```
x = 100

class A:
    def __init__(self, x):
        self.x = x

    def test(self):
        print(x)                    # LEGB
        print(self.x)               # 明确以 self 指定搜索目标
```

```
>>> o = A("abc")
>>> o.test()
100
abc
```

```
>>> dis.dis(o.test)
  8           0 LOAD_GLOBAL              0 (print)
              2 LOAD_GLOBAL              1 (x)            # LEGB
              4 CALL_FUNCTION            1

  9           8 LOAD_GLOBAL              0 (print)
             10 LOAD_FAST                0 (self)         # 先载入 self,
             12 LOAD_ATTR                1 (x)            # 然后为 getattr(self, x)
             14 CALL_FUNCTION            1
```

当两者同时出现时，可能会引发小小的混乱。

前面分析类型创建时提到过，class 在内部以函数方式执行，接收类型名字空间字典作为堆栈帧执行的名字空间。如此一来，成员定义作用域内的 locals 实际指向了 class.__dict__（可读写版）。

但从语法上说，class 是类型定义，而非函数定义，这与内部执行方式无关。因此，class 不构成 E/enclosing 作用域。

```
def enclosing():
    a = "enclosing.a"
```

```
    class A:
        a = "A.a"                               # L --> A.__dict__

        def test(self):
            print("E.a =", a)                   # E --> enclosing.a ③

        print("A.locals =", locals())           # L --> A.__dict__ ①
        print("A.a =", a)                       # L --> A.__dict__ ②


    A().test()
```

```
>>> enclosing()

① A.locals = {'a': 'A.a', 'test': <function A.test>}
② A.a = A.a
③ E.a = enclosing.a
```

7.2 字段

依照所处名字空间的不同，我们将字段分为类型字段和实例字段两类。

> 官方文档将成员统称为 Attribute，不过本书按惯例将数据当作字段。

类型字段在 class 语句块内直接定义，而实例字段必须通过实例引用（self）赋值定义。仅从执行方式来看，无论实例方法存在于哪级类型，其隐式参数 self 总指向当前调用实例。那么通过它创建的字段，也必然存在于该实例的名字空间中。

```
class A:
    def test(self):                         # self 总是引用当前实例，这与继承层次无关
        print(self)
        self.x = 100                        # 向当前实例字段赋值

class B(A):
    pass
```

```
>>> o = B()

>>> hex(id(o))
0x1092244a8

>>> o.test()                              # 观察实例 id
<__main__.B object at 0x1092244a8>
```

> 实例参数 self 只是约定俗成的名字，这类似于其他语言中的 this。

可站在语法和设计角度来看，实例字段乃是所在类型的组成部分，与该类型的相关方法存在关联。所以，我们依然称其为某某类型的实例字段。

字段赋值

可使用赋值语句为类型或实例添加新的字段。

```
>>> class X: pass

>>> X.a = 100                          # 新增类型字段

>>> vars(X)
mappingproxy({'a': 100, ...})
```

```
>>> o = X()
>>> o.b = 200                          # 新增实例字段

>>> vars(o)
{'b': 200}
```

可一旦以子类或实例重新赋值，就将会在其名字空间建立同名字段，并会遮蔽原字段。这与“赋值总是在当前名字空间建立关联”规则一致。

```
>>> class X: pass
>>> o = X()

>>> X.data = 100
>>> o.data                             # 按搜索规则，命中 X.data
```

```
100
```

```
>>> o.data = 200                    # 在 o 名字空间新建 data 实例字段
>>> o.data                          # 按搜索顺序，命中 o.data
200

>>> vars(o)
{'data': 200}
```

```
>>> X.data                          # 不影响原类型字段
100

>>> vars(X)
mappingproxy({'data': 100, ...})
```

删除操作仅针对当前名字空间，而不会按搜索顺序查找类型或基类。

```
>>> class X: pass
>>> X.data = 100

>>> o = X()

>>> del o.data                      # 当前实例名字空间并没有 data 成员
AttributeError: data
```

成员访问总是按搜索规则进行，这与普通变量的静态作用域不同。

这实质上是 LOAD_ATTR 和 LOAD_GLOBAL/LOAD_FAST 的区别。

有关变量静态作用域，请阅读第 4 章。

```
class X:
    data = "X.data"


def test():
    o = X()
    print(o.data)                    # 找到 X.data
```

```
    o.data = "o.data"               # 新增 o.data
    print(o.data)                   # 找到 o.data

    del o.data                      # 删除 o.data
    print(o.data)                   # 找到 X.data
```

```
>>> test()
X.data
o.data
X.data
```

私有字段

将私有字段暴露给用户是很危险的。因为无论是修改还是删除都无法被截获，由此可能引发意外错误。因为语言没有严格意义上的访问权限设置，所以只好将它们“隐藏”起来。

如果成员名字以双下画线开头，但没有以双下画线结尾，那么编译器会自动对其重命名。

```
class X:
    __table = "user"                        # 类型成员

    def __init__(self, name):
        self.__name = name                  # 实例成员

    def get_name(self):
        return self.__name
```

> 同时以双下画线开头和结尾的，通常是系统方法，比如 __init__、__hash__ 等。

所谓重命名，就是被编译器附加了类型名称前缀。虽然这种做法不能真正阻止用户访问，但基于名字的约定也算是一种提示。

```
>>> vars(X)
mappingproxy({'_X__table': 'user', ...})
```

```
>>> vars(X("user1"))
{'_X__name': 'user1'}
```

```
>>> dis.dis(o.get_name)
  6           0 LOAD_FAST                0 (self)
              2 LOAD_ATTR                0 (_X__name)
              4 RETURN_VALUE
```

只是有一个小问题，这种方式让继承类也无法“访问”。

重命名机制总是针对当前类型，继承类型无法访问重命名后的基类成员。可将双下画线前缀改为单下画线，这样虽然不能自动重命名，不过提示作用依旧。

```
class A:
    __name = "user"

class B(A):
    def test(self):
        print(self.__name)             # 被当作当前类型的私有字段重命名: _B__name
```

```
>>> B().test()
AttributeError: 'B' object has no attribute '_B__name'
```

相关函数

下面是几个与类成员有关的内置函数。

```
class A:
    data = "A.data"

class B(A):
    pass
```

```
>>> hasattr(B(), "data")               # 按搜索规则查找整个继承层次
True
```

```
>>> getattr(B(), "data")               # 相当于B().data，同样是按搜索规则进行
```

```
'A.data'

>>> getattr(B(), "name", "abc")          # 返回默认值
'abc'
```

```
>>> o = B()
>>> setattr(o, "data", "o.data")        # 等同于o.data = 'o.data'，针对当前名字空间

>>> o.data
'o.data'
```

```
>>> o = B()
>>> delattr(o, "data")                  # 等同于del o.data，针对当前名字空间
AttributeError: data
```

它们是相关语句的函数版本，可接收动态成员名称，还能用于 lambda 函数。

```
>>> (lambda: getattr(B(), "data"))()
'A.data'
```

7.3 属性

对私有字段会进行重命名保护，那公开字段该如何处理？

问题的核心是访问拦截，必须由内部逻辑决定如何返回结果。而属性（property）机制就是将读、写和删除操作映射到指定的方法调用上，从而实现操作控制。

```
class X:
    def __init__(self, name):
        self.__name = name

    @property                                          # 读
    def name(self):
        return self.__name

    @name.setter                                       # 写（注意语法格式）
```

```
    def name(self, value):
        self.__name = value

    @name.deleter                                          # 删除
    def name(self):
        raise AttributeError("can't delete attribute")
```

```
>>> o = X("user1")
>>> o.name
'user1'

>>> o.name = "abc"
>>> o.name
'abc'

>>> del o.name
AttributeError: can't delete attribute
```

这种@语法被称作装饰器（decorator），详情可参考第 9 章。

多个方法名必须相同。默认从读方法开始定义属性，随后以属性名定义写和删除操作。

如果实现只读，或禁止删除，则只需去掉对应的方法即可。

```
>>> vars(X)
mappingproxy({'name': <property>, ...})
```

当然，也可用 lambda 简化代码。

```
class X:
    name = property(
        fget = lambda self: getattr(self, "_name", None),
        fset = lambda self, value: setattr(self, "_name", value),
    )
```

以方法应对操作，可主动判断是否继续执行。例如，判断赋值数据是否合法，或阻止创建同名实例字段。

```
class X:
    __data = "X.data"

    @property
    def data(self):
        return self.__data

    @data.setter
    def data(self, value):
        if not isinstance(value, str):                    # 检查数据类型
            raise ValueError("value type must be str")

        X.__data = value                                  # 直接修改类型字段
```

```
>>> o = X()

>>> o.data = 1
ValueError: value type must be str

>>> o.data = "abc"
>>> vars(o)
{}
```

属性可实现延迟初始化，无须提前准备数据。且属性方法调用也未必会绑定字段，可按需要从任意数据源获取。

优先级

尽管属性保存于类型名字空间，但其优先级高于同名实例字段。

```
class X:
    data = property(lambda self: "X.data")
```

```
>>> o = X()

>>> o.data = 123                          # 属性优先，data 没有指定赋值方法
AttributeError: can't set attribute

>>> o.__dict__["data"] = 123              # 绕开限制，直接对__dict__操作
>>> vars(o)
```

```
{'data': 123}

>>> o.data                                  # 依旧优先返回 data 属性，而非实例字段
'X.data'
```

> 这种情形是由描述符（descriptor）规则导致的，详情参考第 9 章。

7.4 方法

方法是一种特殊函数，其与特定对象绑定，用来获取或修改对象状态。

实际上，无论是对象构造、初始化、析构，还是运算符，都以方法实现。根据绑定目标和调用方式的不同，方法可分为实例方法、类型方法，以及静态方法。

> 名字以双下画线开始和结束的方法，通常有特殊用途，其由解释器和内部机制调用。

顾名思义，实例方法与实例对象绑定。在其参数列表中，将绑定对象作为第一参数，以便在方法中读取或修改数据状态。在以实例引用调用方法时，无须显式传入第一实参，而由解释器自动完成。

> 官方（PEP8）建议参数名用 self，同样以 cls 作为类型方法的第一参数名。
>
> 当然，也可使用其他命名，解释器并不以参数名字区分方法类别。

```
class X:
    def __init__(self, name):
        self.__name = name

    def get(self):                          # 以实例引用调用，自动传入 self 参数
        return self.__name

    def set(self, value):
        self.__name = value
```

```
>>> o = X("Q.yuhen")

>>> o.get()                              # 忽略第一参数
'Q.yuhen'

>>> o.set("qyuhen")
```

类型方法用来维护类型状态，面向族群提供服务接口。除绑定的第一参数名称不同外，还须添加专门的装饰器，以便解释器将其与实例方法区分开来。

```
class X:
    __data = "X.data"

    @classmethod                          # 定义为类型方法
    def get(cls):                         # 解释器自动将类型对象 X 作为 cls 参数传入
        return cls.__data

    @classmethod
    def set(cls, value):
        cls.__data = value
```

```
>>> X.get()                              # 同样忽略 cls 参数
'X.data'

>>> X.set('hello')
```

至于静态方法，则更像是普通函数。其既不接收实例引用，也不参与类型处理，所以也就没有自动传入的第一参数。使用静态方法，更多的原因是将类型作为一个作用域，或者为当前类型添加便捷接口。

例如，在同一个模块中，分别定义了 AES、DES 等加密类型。那么，基于类型创建静态便捷方法，除避免 aes_encyrpt、des_encrypt 这样的函数命名外，还更利于代码的维护。

```
class DES:
    def __init__(self, key):
        self.__key = key
```

```
    def encrypt_bytes(self, value):                          # 实例方法
        pass

    @staticmethod                                            # 定义为静态方法
    def encrypt(key, s):                                     # 所有参数都须显式传入
        des = DES(key)
        return DES(key).encrypt_bytes(s.encode("utf-8"))
```

```
>>> DES.encrypt("key", "abc")
```

且不论是哪种方法，它们都保存在类型名字空间中，所以不能重名。装饰器和参数的差异，并不能改变在同一名字空间字典中对同一主键重复赋值会覆盖前一次赋值的现实。

```
class X:
    def test(self):
        pass

    @classmethod
    def test(cls, a):
        pass

    @staticmethod
    def test(x):
        pass
```

```
>>> vars(X)
mappingproxy({'test': <staticmethod at 0x1023d5cf8>, ...})
```

绑定

所谓方法绑定的真实面目如何？

```
class X:
    def test(self): pass
```

```
>>> o = X()
>>> o.test
<bound method X.test>

>>> X.test
```

```
<function X.test>
```

同一个方法，仅仅是引用方式不同，结果就有绑定方法和函数的区别。那么奥妙在哪里呢？这其中依然是描述符机制在起作用。它会将实例附加到方法的__self__属性，当解释器执行时就可隐式自动传入 self 参数。

```
>>> o.test.__self__ is o
True

>>> X.test.__self__
AttributeError: 'function' object has no attribute '__self__'
```

当然，就算不是绑定方法，依然可像函数那样调用，无非要显式传入 self 而已。

```
>>> X.test(o)               # 显式传入 self 参数
```

可使用 method.__func__获取__code__的相关信息。

另外，Python 3 已没有 unbound method 的说法，而将其直接当作 function。

自带__self__属性后，无论作为参数传递，还是赋值给变量，总能正确引用原实例。

```
class X:
    def __init__(self, name):
        self.__name = name

    def test(self):
        return self.__name
```

```
>>> a, b = X("a").test, X("b").test         # 将方法赋值给变量

>>> a()
'a'

>>> b()
'b'
```

与之类似的还有类型方法。其实现方式与实例方法完全相同，甚至连存储类型引用的字

段名都还是__self__。至于剩下的静态方法，其不傍他人，总以函数的身份出现。

```
class X:
    @classmethod
    def class_method(cls): pass

    @staticmethod
    def static_method(): pass
```

```
>>> X.class_method.__self__ is X
True

>>> X.static_method
<function X.static_method>
```

特殊方法

下面是几个由解释器自动调用，与对象生命周期相关的方法。

- __new__：构造方法，创建对象实例。
- __init__：初始化方法，设置实例的相关属性。
- __del__：析构方法，实例被回收时调用。

创建实例时，会先后调用构造和初始化方法。

```
class X:
    def __new__(cls, *args):                # 与__init__接收相同的调用参数
        print("__new__", args)
        return super().__new__(cls)

    def __init__(self, *args):              # self 由__new__创建并返回
        print("__init__", args)
```

```
>>> X(1, 2)
__new__  (1, 2)
__init__ (1, 2)
```

如果__new__方法返回的实例与 cls 类型不符，将导致__init__无法执行。

```
class X:
    def __new__(cls):
        print("__new__", cls)
        return [1, 2]                          # 返回结果与 cls 类型不符

    def __init__(self):
        print("__init__")
```

```
>>> X()
__new__ <class 'X'>
```

> 更多详情请参考第 9 章。

7.5 继承

面向对象有 3 个基本特征：封装、继承和多态。

封装讲究结构复用，逻辑内敛，以固定接口对外提供服务。其遵循单一职责，规定每个类型仅有一个引发变化的原因。多于一个的耦合设计会导致脆弱类型，任何职责变更都可能引发连带变故。单一封装的核心是解耦和内聚，这让设计更简单、清晰，代码更易测试和冻结，避免了不确定性。

继承并非要复制原有类型，而是一种增量进化，在遵循原有设计和不改变既有代码的前提下，添加新功能，或改进算法。其对应开闭原则，对修改封闭，对扩展开放。因为修改任何已完成的测试，或正被用户使用的类型，都存在风险。除此之外，还不能违背里氏替换原则，也就是多态特征，所有继承子类应该能直接用于引用父类的场合。保持类型家族整体行为的统一是必要的，诸多设计模式以此为基础，同时保证平滑升级。

继承给设计带来便利的同时，也带来了维护上的麻烦。其天生基于实现的依赖方式导致耦合大范围存在，且过多的继承层次直接增加了代码的复杂度。我们应该分清楚是功能扩展，还是使用某些功能。如果是后者，那么应该以组合来替代。因为组合仅依赖对方的外部接口，没有任何实现上的关联，更无须介入内部运作，很容易替换。当然，组合

无法自动获得祖先能力，需要添加相应的方法才能实现相同的操作接口，更无法直接支持多态。两者非替代关系，都有其适用场景。

另外，我们也习惯于将复杂类型的公用部分剥离出来，形成稳固的抽象基类。其他引发变化的相似因素则被分离成多个子类，以确保单一职责得到遵守，并能相互替换。常见的例子就是类型里的选择分支：先将整体框架抽象成基类，然后每个子类仅保留单一分支逻辑。

封装是将不同部件藏到车壳里，仅向用户提供方向盘、油门、刹车等有限操控接口，通过车载电脑间接控制汽车。

继承则是设计师提供一个通用框架，从技术、生产、维修等整体方向考虑。然后选用不同等级的配件和控制算法，以局部变更快速形成手动、自动，以及舒适和豪华等车型版本。当然，设计师也可以将一个久经考验的成熟车型设计独立出来，以此为基础发展出不同版本。

可无论哪样，每款车都可以车族的身份出场，代表整体的车型设计。但每款车在行驶过程中又能体现出其自身的能力和特点，这便是多态。

在理解面向对象的基本规则后，我们可编写更优雅的代码，进行更自然的设计。

7.5.1 统一类型

Python 3 总算甩掉了陈年包袱，统一了两种互不兼容的类型体系。这让元编程有了更大的发挥余地，也避免了以往需要处理不同类型的麻烦。

Python 2.7

```
>>> class A: pass                   # classic class
>>> class B(object): pass           # new-style class

>>> type(A)
<type 'classobj'>

>>> type(B)
<type 'type'>
```

```
>>> issubclass(A, object)
False

>>> A.__class__
AttributeError: class A has no attribute '__class__'
```

Python 3 不再支持 Classic Class，所有类型都是 New-Style Class。

```
class A    : pass                    # 默认继承自 object
class B(A) : pass
class C(B) : pass
```

```
>>> issubclass(A, object)
True

>>> type(A) is A.__class__
True
```

子类以__base__引用基类，基类也可通过__subclasses__获知所有直接子类。

```
>>> B.__base__
A

>>> A.__subclasses__()                # 仅返回直接子类
[B]
```

其甚至可以介入子类型的创建过程。

__init_subclass__是一个隐式类型方法，在所有层次的子类型创建时被调用，其甚至可接收键值参数。

```
class A:
    def __init_subclass__(cls, **kwargs):
        print("init_subclass:", cls, kwargs)
```

```
>>> class B(A): pass
init_subclass: <class 'B'>  {}
```

```
>>> class C(B, data = "hello"): pass                # 向基类传递参数
init_subclass: <class 'C'>  {'data': 'hello'}
```

7.5.2 初始化

初始化方法__init__是可选的。

如果子类没有新的构造参数，或者新的初始化逻辑，那么没必要创建该方法。因为按照搜索顺序，解释器会找到基类的初始化方法并执行。也因同样的缘故，我们须在子类的初始化方法中显式调用基类方法。

```
class A:
    def __init__(self):
        print("A.init")

class B(A): pass
```

```
>>> B()
A.init
```

可使用名字引用基类方法，或调用 super 返回基类代理，以解除对类型名字的直接依赖。

```
class A:
    def __init__(self, x):
        self.x = x

class B(A):
    def __init__(self, x, y):
        super().__init__(x)
        self.y = y
```

```
>>> o = B(1, 2)

>>> vars(o)
{'x': 1, 'y': 2}
```

无论是 A.__init__，还是 B.__init__，其 self 参数都引用同一实例对象，所以在不同继承层次里创建的实例字段都存储在同一名字空间中，而不会出现多个同名成员。

7.5.3 覆盖

覆盖（override）是指子类重新定义基类方法，从而实现功能变更。在其他某些语言中，仅有虚方法可被覆盖，Python 并无此概念，只要在搜索优先级更高的名字空间中定义同名方法即可。

覆盖不应改变方法参数列表和返回值类型，须确保不影响原有调用代码和相关文档，这与名字遮蔽不同。无论是多态机制，还是名字搜索顺序，都能确保新定义的方法被执行。

```
class A:
    def m(self) : print("A.m")
    def do(self): self.m()              # 如果 self 是 B 类型，那么搜索首先找到的自然是 B.m

class B(A):
    def m(self) : print("B.m")          # override
```

```
>>> A().do()
A.m

>>> B().do()
B.m
```

方法搜索发生在运行期，其不会静态绑定具体类型方法。

```
>>> dis.dis(A.do)
  6           0 LOAD_FAST                0 (self)
              2 LOAD_ATTR                0 (m)            # 运行期搜索 self.m
              4 CALL_FUNCTION            0
```

方法没有属性（property）那样的优先级策略，应注意避免被实例同名成员遮蔽。

7.5.4 多继承

多重继承允许类型有多个继承体系。

这是曾被疯狂追捧，后来被嫌弃的语言特性。从优点上说，它提供了一种混入（mixin）机制，让既有体系的类型轻松扩展出其他体系的功能。但另一方面，它却会导致严重的混乱。试想一个类型同时继承猫和狗会有什么后果？是汪汪叫，还是喵喵叫？这种含糊性直接让设计和代码的复杂度增大到非常麻烦的程度。

新语言大多会避开多继承，改为单继承体系，并严格遵守相关设计原则。至于混入需求，则多以接口实现。如此一来，类型“是什么”是确定的，而类型“能做什么”则以组合来构建。

> 继承表达“是什么”，而接口体现“能做什么”。“是什么”对接口来说并不重要，它也不关心起源和后续。

作为“老古董”，Python 依然保留对多继承的支持。不过大家使用时须慎重，能免则免。

```
class A:
    def a(self): pass

class B:
    def b(self): pass

class X(A, B): pass
```

```
>>> dir(X)                              # 可访问所有基类成员
['a', 'b']
```

要获取全部基类，可通过__bases__属性。

```
>>> X.__base__
A
```

```
>>> X.__bases__
(A, B)
```

不同于__base__仅返回单个基类的只读属性，__bases__按继承顺序返回全部基类。它还能用来调整基类顺序，或动态插入新的基类，实现功能混入。

```
class C:
    def c(self): pass
```

```
>>> X.__bases__ = (B, A, C)            # 调整继承顺序，并添加新的基类

>>> dir(X)                             # 可访问新基类成员
['a', 'b', 'c']
```

除横向动态扩展外，其还可纵向注入继承体系中。

```
class A:
    def test(self): print("A.test")

class B(A):
    pass
```

```
class Proxy(A):
    def test(self):                       # 拦截 A.test 调用
        print("proxy")
        return super().test()
```

```
>>> B.__bases__ = (Proxy,)              # 修改 B 基类，形成 B -> Proxy -> A 方式

>>> B().test()
proxy
A.test
```

__mro__

任何类型的最终基类都是 object，所以多重继承会出现菱形布局。可最大的问题是，当多条继承线上出现相同名字的成员时，如何选择？

这涉及成员搜索的基本规则：MRO（method resolution order）。对于单继承，由近到远向祖先类依次查找。而复杂性在于，多继承在深度和广度间的排列。

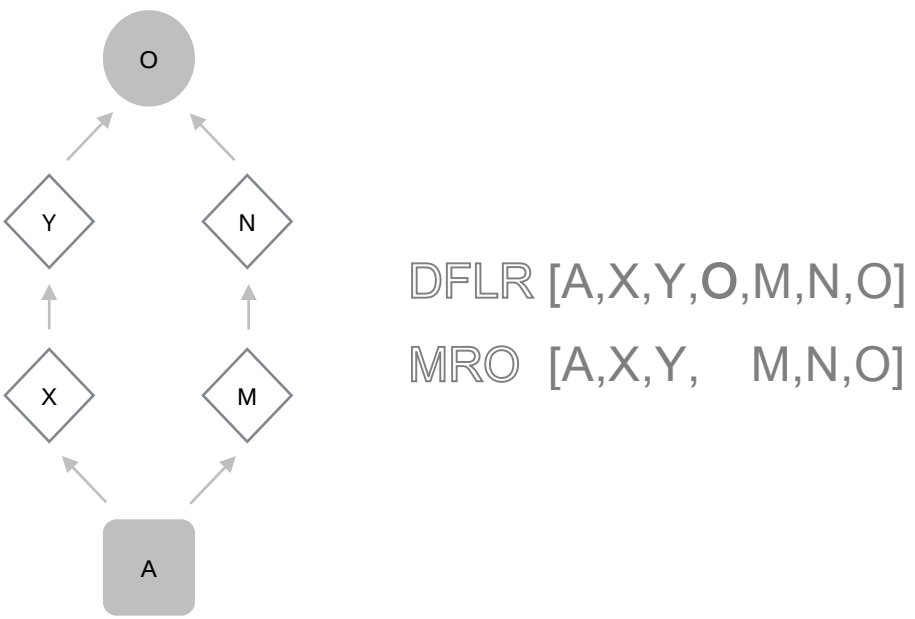
> 在 Python 2 时期，Classic Class 使用深度优先的 DFLR（depth-first, left-to-right）模式。
>
> 后来 New-Style Class 因单根继承的关系，进化为 MRO（breadth-first than depth-first）算法。

MRO 步骤如下：

1．按“深度优先，从左到右”顺序获取类型列表。

2．移除列表中的重复类型，仅保留最后一个。

3．确保子类总在基类前，并保留多继承定义顺序。




```
class Y:
    n = "Y.n"
    def test(self): print("Y.test")
```

```
class X(Y): pass
```

```
class N: pass

class M(N):
    n = "M.n"
    def test(self): print("M.test")
```

```
class A(X, M): pass

>>> A.__mro__
(A, X, Y, M, N, object)
```

```
>>> A().n
'Y.n'

>>> A().test()
Y.test
```

> 简单一点说，就是“左侧优先，子类优先”。当然，这不违背实例优先级最高的原则。
>
> 另外，__mro__是只读属性，且不能通过实例访问。

按算法规则，对__bases__的调整会直接影响__mro__的结果。

```
>>> A.__bases__ = (M, X)

>>> A.__mro__
(A, M, N, X, Y, object)

>>> A().test()
M.test
```

任何时候，都应避免多条继承线存在交叉的现象，并竭力控制多继承和继承深度。这有助于梳理其相互关系，为代码可读和可维护性提供保障。

super

该函数返回基类型代理，完成对基类成员的委托访问。

这里有两个参数，第二参数对象提供__mro__列表，第一参数则指定起点。函数总是返回起点位置后一类型。这要求第二参数必须是第一参数的实例或子类型，否则第一参数不会出现在列表中。同时，第二参数还是实例和类型方法所需的绑定对象。

算法伪码

```
def super(t, o):
    mro = getattr(o, "__mro__", type(o).__mro__)        # 返回实例或类型的 mro 列表
    index = mro.index(t) + 1                            # 返回列表中的下一类型
    return mro[index]
```

对单继承，通常省略参数，默认使用当前类型和实例，返回直接基类。但对多继承，则须明确指定起点和列表提供者。

```
class A:
    @classmethod
    def demo(cls): pass
    def test(self): print("A")

class B(A):
    def test(self): print("B")

class C(B):
    def test(self): print("C")
```

```
>>> C.__mro__
```

```
(C, B, A)

>>> super(C, C).test
<function B.test>

>>> super(B, C).test
<function A.test>
```

依旧使用__self__作为绑定属性，存储第二参数引用。

```
>>> o = C()

>>> super(C, o).__self__ is o
True
```

```
>>> super(C, o).test
<bound method B.test>

>>> super(C, C).demo
<bound method A.demo>
```

不建议直接使用__class__.__base__访问基类，这可能引发一些意外错误。

```
class A:
    def test(self):
        print("A.test")

class B(A):
    def test(self):
        print("B.test")
        self.__class__.__base__.test(self)   # 站在 B 的角度，self.__class__.__base__似乎指向 A
                                             # 但实际上，self 可能是 B 的子类实例，比如 C
                                             # 如此，self.__class__.__base__实际指向的依然是 B
                                             # 其结果就是导致 B.test 被递归
class C(B): pass
```

```
>>> C().test()
RecursionError: maximum recursion depth exceeded in comparison
```

这里的正确方法是采用显式类型名字访问，或用 super 函数。即便省略 super 参数，它默认也使用当前类型 B 作为搜索起点。这样就可确保访问 A.test 方法，避免递归错误。

7.5.5 抽象类

抽象类表示部分完成，且不能被实例化的类型。

作为一种设计方式，抽象类可用来分离主体框架和局部实现，或将共用和定制解耦。不同于接口纯粹的调用声明，抽象类属于继承树的组成部分，其允许有实现代码。从抽象类继承，必须实现所有层级未被实现的抽象方法，否则无法创建实例。

抽象类就像一个准系统。汽车厂家在完成整体设计和主要部件生产后，再由不同地区的分公司依照当地路面、油料以及法规选用不同配件，完成适合当地销售策略的成品。

定义抽象类，须继承 ABC，或使用 ABCMeta 元类。

```
from abc import ABCMeta, abstractmethod

class Store(metaclass = ABCMeta):

    def __init__(self, name):
        self.name = name

    def save(self, data):
        with open(self.name, "wb") as f:
            d = self.dump(data)
            f.write(d)

    def read(self):
        with open(self.name, "rb") as f:
            return self.load(f.read())

    @abstractmethod
    def dump(self, data): ...

    @abstractmethod
```

```
    def load(self, data): ...
```

```
>>> Store("test.dat")
TypeError: Can't instantiate abstract class Store with abstract methods dump, load
```

该示例定义存储框架，并实现了通用部分的接口方法。至于具体的序列化过程，则交由子类完成。如此，可按需创建不同版本。

```
import pickle

class PickleStore(Store):

    def dump(self, data):
        return pickle.dumps(data)

    def load(self, data):
        return pickle.loads(data)
```

```
>>> s = PickleStore("test.dat")

>>> s.save({"x":1, "y":2})

>>> s.read()
{'x': 1, 'y': 2}
```

如果从抽象类继承，未实现全部方法，或添加新的抽象定义，那么该类型依然是抽象类。另外，抽象装饰器还可用于属性、类型方法和静态方法。

```
from abc import ABCMeta, abstractmethod

class A(metaclass = ABCMeta):

    @classmethod                                # 注意两个装饰器的顺序
    @abstractmethod
    def hello(cls): ...

class B(A): pass
```

```
>>> B()
```

```
TypeError: Can't instantiate abstract class B with abstract methods hello
```

即便是抽象类型方法，依然需要实现，否则无法创建实例。

```
class B(A):
    @classmethod
    def hello(cls): pass
```

```
>>> B()
<B at 0x105021940>
```

7.6 开放类

在运行期间，可动态向实例或类型添加新成员，其中也包括方法。

有别于实例字段，方法需要添加到类型名字空间，因为要作用于所有实例和后续类型。另外，只有添加到类型名字空间才能自动构成绑定，这是由解释器的运行规则所决定的。

> 将已绑定的方法添加到实例名字空间，只能算是字段赋值，算不上添加方法。
>
> 而将一个函数添加到实例，即便手工设定__self__也无法构成绑定。

```
>>> class X: pass

>>> o = X()
>>> o.test = lambda self: None                # 向实例添加函数字段
>>> o.test.__self__ = o                       # 尝试手工设定绑定

>>> o.test                                    # 依然无法构成绑定，除非重写描述符的相关方法
<function <lambda>>

>>> o.test()
TypeError: <lambda>() missing 1 required positional argument: 'self'
```

考虑到类型__dict__是一个只读代理，其只能以字段赋值方式添加新的方法。如果是类

型和静态方法，还须调用装饰器函数。

> 受益于动态成员查找方式，新增的方法对已创建的实例同样有效。

```
>>> class X: pass
>>> o = X()                                    # 预先创建的实例
```

```
>>> X.a = lambda self: print(f"instance method: {self}")
>>> X.b = classmethod(lambda cls: print(f"class method: {cls}"))
>>> X.c = staticmethod(lambda: print("static method"))
```

```
>>> o.a()
instance method: <X object at 0x111688908>

>>> o.b()
class method: <class 'X'>

>>> o.c()
static method
```

object

作为祖先根类的 object，有点特殊，我们无法向其类型和实例添加任何成员。

```
>>> object.x = 1
TypeError: can't set attributes of built-in/extension type 'object'

>>> o = object()
>>> o.x = 1
AttributeError: 'object' object has no attribute 'x'
```

实例甚至没有 __dict__ 属性。

```
>>> object().__dict__
AttributeError: 'object' object has no attribute '__dict__'
```

对此，可用 SimpleNamespace 替代它。

SimpleNamespace 简单继承自 object，用于替代那些“class X: pass”语句。

不同于 namedtuple 创建结构化类型，SimpleNamespace 直接创建实例。

```
>>> o = types.SimpleNamespace(a = 1, b = "abc")
>>> o.c = [1, 2]
>>> o.__dict__
{'a': 1, 'b': 'abc', 'c': [1, 2]}
```

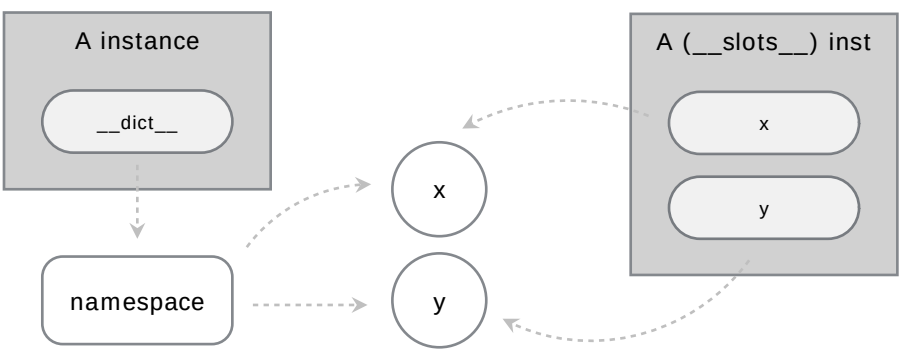
__slots__

名字空间字典带来便利的同时，也会造成内存和性能问题。

对那些需要创建海量实例的类型，须做“固化”处理。通过设置__slots__，阻止实例创建__dict__等成员。解释器仅为指定成员分配空间，添加任何非预置属性都会引发异常。

对于有__slots__设置的类型，解释器在创建类型对象时，直接将指定成员包装成描述符后静态分配到类型对象尾部。实例字段也不再通过__dict__存储，而是改为直接分配引用内存。这样一来，只能替换引用内容，无法改变成员名字，更无法分配新的成员空间。

成员后续操作均由描述符完成。可间接修改或删除字段内容，但却无法改变描述符本身。




```
class A:
    __slots__ = ("x", "y")
```

```
>>> o = A()

>>> o.z = 100                                          # 不能新增属性，因为无法为其分配引用内存
AttributeError: 'A' object has no attribute 'z'

>>> o.x = 1                                            # 只能为预定字段赋值
>>> o.y = 2
```

```
>>> o.x = "abc"                                        # 通过描述符修改字段内容
>>> o.x
'abc'
```

```
>>> del o.x                                            # 删除字段内容，但描述符依然存在
>>> o.x = 100                                          # 引用内存也在，可重新赋值
```

对__slots__的修改并不会影响类型创建时设定的内存分配策略，加上预定成员以描述符实现，所以也无法更换其名称。

```
>>> A.__slots__ = ("x", "y", "z")

>>> o = A(1, 2)
>>> o.z = 3
AttributeError: 'A' object has no attribute 'z'
```

```
>>> A.__slots__ = ("a", "b")

>>> dir(A)
['x', 'y']
```

> 注意，__slots__限制的是实例，而非类型。因此，仍然可以为类型添加新成员。
>
> 如果在__slots__中添加__dict__，可回归其“原来”的样子，不过那就没什么意义了。

继承有__slots__设置的类型，同样需要添加该设置。其可为空，或是新增字段，以指示解释器继续使用特定分配策略。

```
class A:
    __slots__ = ("x", "y")

    def __init__(self, x, y):
        self.x = x
        self.y = y
```

```
class X(A):                                    # 没有__slots__，会创建__dict__

    def __init__(self, x, y, z):
        super().__init__(x, y)
        self.z = z


class Y(A):
    __slots__ = ("z",)                         # 新增字段列表，或为空

    def __init__(self, x, y, z):
        super().__init__(x, y)
        self.z = z
```

```
>>> x = X(1, 2, 3)

>>> x.__dict__                                 # 包含__dict__，用于存储非__slots__新增成员
{'z': 3}
```

```
>>> y = Y(1, 2, 3)

>>> y.__dict__
AttributeError: 'Y' object has no attribute '__dict__'

>>> dir(y)
['x', 'y', 'z']
```

在了解了功能细节后，开始测试我们所关注的内存和性能问题。在此先用 pympler 详细统计实例及其成员的全部内存开销，以做初步对比。

```
class A:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

```
class B:
    __slots__ = ("x", "y")

    def __init__(self, x, y):
        self.x = x
        self.y = y
```

从下面的输出结果来看，__dict__引发了“巨大”的内存开销。这对单个或少量实例或许毫不起眼，可其一旦海量叠加就是大麻烦。

```
>>> from pympler.asizeof import asized
```

```
>>> print(asized(A(1, 2), detail = 2).format())

<A> size=344 flat=56
    __dict__ size=288 flat=112
        [K] x    size=56 flat=56
        [K] y    size=56 flat=56
        [V] x: 1 size=32 flat=32
        [V] y: 2 size=32 flat=32
    __class__ size=0 flat=0
```

```
>>> print(asized(B(1, 2), detail = 2).format())

<B> size=192 flat=56
    __slots__ size=72 flat=72
            x size=32 flat=32
            y size=32 flat=32
    __class__ size=0  flat=0
```

继续模拟百万级别的“海量”需求，观察创建实例所需的内存和时间开销。

test.py

```
class A:
    def __init__(self):
        self.data = 100
```

```
class B:
    __slots__ = ("data")

    def __init__(self):
        self.data = 100


@profile
def test():
    x = list(range(1000000))
    a = [A() for i in x]
    b = [B() for i in x]


if __name__ == '__main__':
    test()
```

其创建时间的差距不算太大，姑且标记为中低优先级。但其内存开销却相差 3 倍，亟待解决。毕竟，相比于大多数的真实案例，百万级别真算不上什么大数字。

```
$ kernprof -l -v test.py

Hits         Time    Per Hit    % Time      Line Contents
==============================================================
   1      1389059   1389059.0     53.9      a = [A() for i in x]
   1      1153548   1153548.0     44.7      b = [B() for i in x]
```

```
$ python -m memory_profiler test.py

  Mem usage      Increment       Line Contents
================================================
233.246 MiB    161.895 MiB       a = [A() for i in x]
287.824 MiB     54.578 MiB       b = [B() for i in x]
```

测试结果仅供参考，请按实际需求组织测试用例。

测试工具的使用方法请参考第 11 章。

7.7 运算符重载

每种运算符都对应一个有特殊名称的方法。

> 解释器会将运算符指令转换成方法调用，方法名可参考标准库的 operator 文档。
>
> 运算符重载就是定义目标方法。但不限于运算符，还包括内置函数和某些语句。

```
class X:
    def __init__(self, data):
        self.data = data

    def __repr__(self):
        return f"{self.__class__} {self.data!r}"

    def __add__(self, other):                          # 加法+
        return X(self.data + other.data)

    def __iadd__(self, other):                         # 增量赋值+=
        self.data.extend(other.data)
        return self
```

```
>>> a, b = X([1, 2]), X([3, 4])

>>> a + b
<class 'X'> [1, 2, 3, 4]

>>> a += b
>>> a
<class 'X'> [1, 2, 3, 4]
```

__repr__

函数 repr、str 都用于输出对象字符串格式信息。

方法__repr__倾向于输出运行期状态，比如类型、id，以及关键性内容，其适合调试和观察。而__str__通常返回内容数据，其面向用户，易阅读，可输出到终端或日志。

```
class X:
    def __init__(self, x):
        self._x = x

    def __repr__(self):
        return f"<X:{hex(id(self))}>"

    def __str__(self):
        return str(self._x)
```

```
>>> repr(X(1))
'<X:0x10664d518>'

>>> str(X(1))
'1'
```

__item__

为对象添加索引（index）或主键（key）访问。

```
class X:
    def __init__(self, data = None):
        self.data = data or []

    def __getitem__(self, index):
        return self.data[index]

    def __setitem__(self, index, value):
        if index >= len(self.data):
            self.data.append(value)
            return

        self.data[index] = value

    def __delitem__(self, index):
        return self.data.pop(index)
```

```
>>> o = X([1, 2, 3, 4])

>>> o[2] = 300
```

```
>>> del o[2]
>>> o.data
[1, 2, 4]
```

__call__

让对象可以像函数那样调用。

> 实现该方法的对象被称作 callable，函数也是其中的一种。
>
> 用实例对象替代函数，让逻辑自带内部状态（其封装性比闭包更好）。

```
class X:
    def __init__(self):
        self.count = 0

    def __call__(self, s):
        self.count += 1
        print(s)


def test(f):
    f("abc")
```

```
>>> test(lambda s: print(s))
abc

>>> test(X())
abc

>>> o = X()
>>> o("abc")
abc
```

__dir__

定制 dir 函数的返回值，隐藏部分成员。

```
class X:
```

```
    def __dir__(self):
        return (m for m in vars(self).keys() if not m.startswith("__"))
```

```
>>> o = X()
>>> o.a = 1
>>> o.b = 2

>>> dir(o)
['a', 'b']
```

__getattr__

下面介绍几个与实例属性访问相关的方法。

- __getattr__：当整个搜索路径都找不到目标属性时触发。
- __setattr__：拦截对任何属性的赋值操作。
- __delattr__：拦截对任何属性的删除操作。

拦截到赋值和删除操作后，由拦截方法负责处理赋值和删除行为，忽略则被视为放弃该操作。

在拦截方法内部，通过属性或 setattr 等函数调用都可能再次被拦截，甚至引发递归调用错误。应直接操作__dict__，或使用基类 object.__setattr__方法。

```
class A:
    a = 1234


class B(A):
    def __init__(self, x):
        self.x = x

    def __getattr__(self, name):
        print(f"get: {name}")
        return self.__dict__.get(name)

    def __setattr__(self, name, value):
        print(f"set: {name} = {value}")
        self.__dict__[name] = value
```

```
    def __delattr__(self, name):
        print(f"del: {name}")
        self.__dict__.pop(name, None)
```

```
>>> o = B(1)
set: x = 1                              # 拦截B.__init__里对self.x的赋值操作

>>> o.a                                 # 搜索路径里能找到的成员，不会触发__getattr__
1234

>>> o.xxx                               # 找不到，才会触发
get: xxx

>>> o.x = 100                           # 任何赋值操作都会被__setattr__拦截（无论其是否存在）
set: x = 100

>>> del o.x                             # 拦截任何删除操作,（无论其是否存在）
del: x
```

方法__getattribute__拦截任何实例属性的访问（无论其是否存在）。

拦截目标包括__dict__，这意味着只能通过基类方法进行操作。

访问不存在的成员时，__getattribute__拦截后不再触发__getattr__，除非显式调用或触发异常。

在下面的实例中，因 object.__getattribute__找不到目标属性，所以会触发 A.__getattr__调用，继而还会拦截到其内部对__dict__的访问。

```
class A:
    def __init__(self, x):
        self.x = x

    def __getattr__(self, name):
        print(f"getattr: {name}")
        return self.__dict__.get(name)

    def __getattribute__(self, name):               # 返回结果，或引发AttributeError
        print(f"getattribute: {name}")
        return object.__getattribute__(self, name)
```

```
>>> o = A(1)

>>> o.x                              # 无论其是否存在，均被拦截
getattribute: x
1

>>> o.s
getattribute: s                      # __getattribute__拦截
getattr: s                           # object.__getattribute__找不到 s，触发__getattr__
getattribute: __dict__               # __getattr__访问__dict__被再次拦截
```

第 8 章　异常

8.1　异常概述

当程序执行出现错误，或返回非预期结果时，通常会抛出一个异常对象用来表达非正常状态。但我们不能因此就认为异常等同于错误，因为异常只是一种运行期状态值。

常见的错误处理方式有两类：一类是函数返回错误码，由调用方决定如何处理；另一类是以专用语句保护代码块，当异常发生时，跳转到指定的处理单元。前者有较好的性能，而后者可分离正常逻辑与故障处理代码。

将错误码作为函数签名的组成部分，可让调用方预先知道可能会发生什么。其常见于系统编程中，尤其是系统调用函数中，因为错误码本身就是结果的预选项之一。且该机制无须处理调用堆栈信息，无须额外状态保存和流程控制机制，其简单、高效。但也正因为其执行流程不会主动中断，所以必须检查全部返回值，任何忽略行为都可能引发连续错误。

这对应用开发是很难忍受的，逻辑处理中充斥着冗长的错误处理代码，这直接导致代码的可读性变差，更不便于维护和重构。相比而言，结构化异常要更优雅一些。虽然其需要付出额外的运行期代价，但却可以统一处理，毕竟错误处理仅是意外分支。当然，对异常的捕获同样需要倍加小心，任何隐式行为都存在风险。

执行机制

在系统内部，解释器使用一种被称作“块栈”（block stack）的结构处理异常逻辑。它和执行栈一起被栈帧管理。这里的区别在于，块栈专门用来处理循环和异常跳转。在运行期，相关指令会提前将跳转位置信息存储到块栈，需要时直接从中获取即可。

```
def test():
    try:
        raise Exception
    except:
        print("except")
```

```
>>> dis.dis(test)
      0 SETUP_EXCEPT             8 (to 10)          # 向块栈添加 except 跳转位置设置

      2 LOAD_GLOBAL              0 (Exception)      # 创建并引发异常
      4 RAISE_VARARGS            1                  # 解释器从块栈弹出设置，按参数跳转

      6 ...
      8 ...

>>   10 POP_TOP                                     # 跳转到此处
     12 POP_TOP                                     # 清除 exc 参数
     14 POP_TOP

     16 LOAD_GLOBAL              1 (print)          # 执行 except 代码
     18 LOAD_CONST               1 ('except')
     20 CALL_FUNCTION            1
     22 POP_TOP
     24 POP_EXCEPT                                  # 移除块栈内的设置
     26 JUMP_FORWARD             2 (to 30)          # 处理后，函数正常返回
     28 END_FINALLY

>>   30 LOAD_CONST               0 (None)
     32 RETURN_VALUE
```

无论异常是主动抛出，还是函数调用引发，都会导致解释器转入错误处理逻辑。解释器会检查当前块栈内是否有匹配的处理逻辑，如果有则跳转并执行相应的指令。如果没有，那么当前函数被终止，并返回 NULL。该返回值会导致上一级函数再次进入错误处

理逻辑。其结果是要么异常被捕获；要么沿调用堆栈向外传递，直至进程崩溃。

解释器执行伪码

```
EvalFrame (Frame *f)
{
    for(;;)
    {
        RAISE_VARARGS:
            do_raise()                      // 保存异常对象
            goto error

        CALL_FUNCTION:
            res = call_function()
            if (res == NULL) goto error     // 如果调用的函数返回 NULL，则表示有错误发生

        RETURN_VALUE:
            retval = POP()
            why = WHY_RETURN                // 正常结束标志
            goto end

error:
        why = WHY_EXCEPTION                 // 异常发生标志

end:
        /* 如果有异常发生，则检查当前块栈是否有匹配处理逻辑
         * 如果有对应的设置，那么跳转执行处理指令
         * 被捕获处理后，必然以 RETURN_VALUE 指令结束，从而解除异常
         * 鉴于可能有多层嵌套，需要循环检查。*/
        while (why != WHY_NOT && iblock > 0)
        {
            // 从块栈弹出设置
            b = f->block[iblock - 1]
            f->iblock--

            // 跳转执行
            if (why == WHY_EXCEPTION && (SETUP_EXCEPT || SETUP_FINALLY)) {
                why = WHY_NOT
                JUMPTO(handler)
                break
            }
        }
```

```
            // 如果异常未被处理，则终止当前函数的执行
            if (why != WHY_NOT) break
        }

        // 非正常结束，函数返回 NULL
        if (why != WHY_RETURN) retval = NULL

    exit:
        /* 终止当前函数的执行，回到上一级函数调用
         * 如异常未被处理，那么上级函数的 CALL_FUNCTION 就返回 NULL，这导致再次进入 error 逻辑。*/
        tstate->frame = f->f_back
        return retval
    }
```

> 注意：NULL != None。函数默认返回的 None 是正常对象，有值，有地址。

异常对象被保存到当前线程状态里，其可用 sys.exc_info 获取。

```
def test():
    try:
        raise Exception("err")
    except:
        print(sys.exc_info())
```

```
>>> test()
(<class 'Exception'>, Exception('err',), <traceback object at 0x106c6b248>)
```

用线程保存异常对象，仅能确保外层调用可获取异常状态，但依旧缺少一份重要数据：这就是在什么地方，因为什么原因导致异常发生，这对代码调试尤为重要。为此，当异常发生时，解释器会逐级为每个未能捕获异常的函数调用创建关联跟踪对象（traceback），它负责存储栈帧、最后指令的位置，以及源码行等信息。如此一来，就算引发异常的函数结束，在外层调用中依然可以查询到详细的执行信息。

与栈帧类似，对应函数的跟踪对象也构成链表结构，这样就能还原异常发生时的完整调用堆栈状态。其可用来恢复现场，进入调试模式，或输出相关数据。

```
def div(a, b):
    return a/b

def main():
    try:
        print(div(10, 0))
    except:
        _, _, tb = sys.exc_info()
        print(traceback.print_tb(tb))
```

```
>>> main()
File "<python>", line 6, in main
  print(div(10, 0))
File "<python>", line 2, in div
  return a/b
```

异常一旦被捕获处理，保存在线程状态内的 exc_type、exc_value、exc_traceback 都会被清除。

异常处理

完整的异常处理结构包含 4 个组成单元。

- try：需要保护的代码块。
- except：异常发生时，按所属类型捕获并处理。
- else：异常未发生时执行，但最少有一个 except 语句。
- finally：无论异常是否发生，总是执行。

后 3 个处理单元可选，但必须按顺序排列，否则会导致语法错误。

```
def test(n):
    try:
        print("try")
        if not n: raise Exception
    except:
        print("except")
    else:
        print("else")
```

```
    finally:
        print("finally")
```

```
>>> test(1)
try
else
finally
```

```
>>> test(0)
try
except
finally
```

当被保护代码块发生异常时，会终止其执行流程，并立即跳转到相关异常处理单元。即便异常被捕获并修复，也不再恢复原有流程，而是从处理单元后继续执行。如此，我们应该细心选择保护范围。

```
def test():
    try:
        print("try")
        raise Exception
        print("try2")                  # 此行代码不会被执行
    except:
        print("except")

    print("exit")                      # 异常处理完成后，继续后续代码的执行
```

```
>>> test()
try
except
exit
```

使用 raise 语句主动抛出异常，但必须是 BaseException 子类或实例。

```
>>> raise "abc"
TypeError: exceptions must derive from BaseException
```

> 通常选择 Exception，可直接使用类型，由解释器自动创建实例。

多个 except 语句按顺序匹配，所以应像选择语句那样，优先处理更具体的异常类型。除非必要，不建议捕获全部异常（Exception）。首先，可能隐式忽略需要修复的错误；其次，意外拦截其他用途的异常信息。

```
def test():
    try:
        raise IndexError
    except Exception:                           # 按顺序匹配，所以在此处捕获
        print("Exception")
    except IndexError:                          # 永远不会被执行，与原意不符
        print("IndexError")
```

```
>>> test()
Exception
```

可同时匹配多种类型，并导出异常对象实例。

```
def test():
    try:
        raise KeyError("key")
    except (IndexError, KeyError) as exc:       # 与 sys.exc_info 返回相同的实例
        print(repr(exc), exc.__traceback__)     # 输出异常及其跟踪对象信息
```

```
>>> test()
KeyError('key') <traceback object at 0x102ed6248>
```

> 注意：Python 3 不再支持 Python 2 的某些语法。

被捕获的异常可原样重新抛出。比如说仅记录不处理，或者将异常作为广播消息继续向外传播，等等。当然，也可抛出新异常对象。

```
def div(a, b):
    return a/b

def test():
    try:
        div(10, 0)
    except Exception as exc:
```

```
        print(f"[log] {exc}")              # 仅做日志，不做处理
        raise                              # 原样抛出

def main():
    try:
        test()
    except:
        _, val, tb = sys.exc_info()        # 重新抛出的异常，不会改变跟踪信息
        print(f"[exc] {val}")
        traceback.print_tb(tb)
```

```
>>> main()
[log] division by zero
[exc] division by zero

  File "<python>", line 21, in main
    test()
  File "<python>", line 13, in test
    div(10, 0)
  File "<python>", line 8, in div
    return a/b
```

所导出的异常实例名字在语句块结束时，会被自动删除。这与普通名字作用域不同。

Python 2 不会删除所导出的异常实例名字，但可能会导致循环引用、垃圾回收延迟等负面效应。

```
def test():
    try:
        x = 1                              # 整个函数作用域
        raise Exception("err")
    except Exception as exc:               # 异常变量 exc 仅在当前 except 内有效
        print("except:", locals())
    finally:
        print("finally:", locals())

    print("function:", locals())
```

```
>>> test()
except   : {'exc': Exception('err',), 'x': 1}
finally  : {'x': 1}
```

```
function : {'x': 1}
```

因 finally 总被执行，所以常用来执行清理操作，确保相关资源被及时回收。

```
def test():
    try:
        raise Exception("error")
    finally:                                # 即便异常发生，也会先执行 finally 语句
        print("dispose")

def main():
    try:
        test()
    except Exception as exc:                # 随后，才是外层拦截捕获
        print(exc)
```

```
>>> test()
dispose
error
```

> 即便在 try 语句块内执行 return、break 等逻辑跳转指令，解释器也会确保 finally 被执行。

当然，如果 try 语句未被执行，那么 finally 也同样不会被执行，它俩是一个整体。

```
def test(n):
    if not n: return

    try:
        print("try")
        return
    finally:
        print("finally")
```

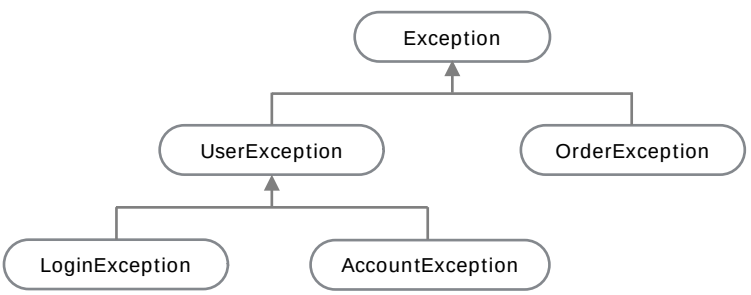
```
>>> test(1)
try
finally

>>> test(0)
```

自定义类型

鉴于 except 语句以类型作为匹配条件，因此我们应该有针对性地抛出具体的异常类型，而非用字符串或错误码进行区分。

> 异常对象内的字符串或错误码属性通常作为日志，或便于阅读的附加内容。
>
> 在某些规范中，建议异常字符串不包含结束标点，以便嵌入其他格式化文本中。

使用自定义异常类型，可以存储更多的格式化信息用于诊断和修复。或者按逻辑分类构建异常类型树，以便通过不同基类以不同粒度捕获异常。同时，其与逻辑扩展相对应，派生出更多详细类型。



自定义异常类型直接或间接继承自 Exception，通常以 Error 和 Exception 为名称后缀。

> 内置异常多以 Error 结尾，但建议以 Exception、Error 后缀区分可修复异常和不可修复错误。
>
> 日志和显示用信息依旧使用 Exception.args，以便通用模块在不了解异常对象结构的情况下，记录和输出可阅读信息。但用于诊断和修复的数据，建议使用显式属性存储。尽管 Exception.args 也能存储多个数据，但会造成处理逻辑依赖参数顺序和索引序号，不如显式属性合理。

```
class UserException(Exception):
    def __init__(self, uid):
        self.uid = uid                                          # 用于用户修复逻辑
        super().__init__(f"an exception about user {uid}", uid)     # 用于日志和显示
```

```
def test():
    try:
        raise UserException(1001)
    except UserException as exc:
        logging.error(exc.args)                                    # 日志记录
        print(exc.uid)                                             # 修复或重新抛出
```

```
>>> test()
ERROR: ('an exception about user 1001', 1001)
```

异常链

使用 raise 抛出异常时，还可用 from 子句接收另一异常，形成链式结构。但这不意味着会引发多次异常处理，每个线程仅有最后一个异常可被捕获。其最大用途在于，在保留原异常状态的情况下，包装并抛出新的异常。

```
def test():
    try:
        raise Exception("err")
    except Exception as exc:
        raise Exception("wrap") from exc              # 使用链保留原异常状态

def main():
    try:
        test()
    except Exception as exc:
        while True:                                   # 循环访问链上的所有异常
            print(repr(exc), exc.__traceback__)
            if not exc.__cause__: break
            exc = exc.__cause__
```

```
>>> main()
Exception('wrap',) <traceback object at 0x102ec9e48>
Exception('err',) <traceback object at 0x102ec9b48>
```

反过来，我们也可以主动打断异常链构造。

```
def test():
    try:
```

```
        try:
            raise Exception("bbb") from Exception("aaa")
        except Exception as exc:
            raise exc from None                          # 打断异常链
    except Exception as exc:
        print(repr(exc))
        print(repr(exc.__cause__))
```

```
>>> test()
Exception('bbb',)
None
```

实际上，即便不构造异常链，解释器也会以__context__保存前一个异常对象。只是不能像 from 语句那样，主动构造和选择目标。

```
def test():
    try:
        try:
            raise Exception("err")
        except Exception as exc:
            raise Exception("wrap")          # 抛出新异常时，自动使用__context__保存前异常对象
    except Exception as exc:
        print(repr(exc.__context__))
```

```
>>> test()
Exception('err',)
```

如果异常未被捕获，解释器会输出__cause__、__context__里的所有异常信息。

使用模式

除用于错误和非正常状态处理外，异常还可以用来控制执行流程。比如，从多重嵌套循环中跳出，或终止整个调用堆栈。

```
def test():
    class LabelException(Exception): pass

    try:
```

```
            while True:
                while True:
                    while True:
                        print("loop")
                        raise LabelException
        except LabelException:
            pass

        print("exit")
```

```
>>> test()
loop
exit
```

定义多个不同 LabelException 类型，然后在不同地方捕获，这样就可以实现多个跳转标签。

不同于 return 只能终止当前函数，异常会沿调用堆栈向外传递。只需在合适的地点拦截，即可终止多级函数调用过程。这也是不建议拦截全部异常的理由之一。

test.py

```
class CancelException(Exception): pass

def db():
    raise CancelException

def logic():
    print("logic start")
    db()
    print("logic end")                    # 不会执行

def request():
    print("request start")
    logic()
    print("request end")                  # 不会执行

def main():
    try:
        request()
    except CancelException:
        print("request cancel")
```

```
if __name__ == '__main__':
    main()
```

```
$ python test.py

request start
logic   start
request cancel
```

还可以用来插入调试干预代码。

```
def div(x, y):
    z = x / y
    return z

def test():
    try:
        x, y = 10, 0
        div(x, y)
    except Exception:                          # 如果出现未被处理的异常，则启动调试器
        if __debug__: pdb.post_mortem()
```

```
>>> test()
(pdb) w
  <python>(8) test()
-> div(x, y)
> <python>(2) div()
-> z = x / y

(pdb) locals()
{'y': 0, 'x': 10}
```

除此之外，我们还可以向生成器对象发送异常，作为信号干预其执行流程。相关内容请参考前面的章节，此处不再赘述。

警告提示

标准库另有一个 warnings 模块，用于输出警告信息。比如，提示开发人员当前算法已过

时，或当前为测试版等。默认输出不会干扰程序的执行，但可配置忽略，或转换为异常。

test.py

```
import warnings
class BetaWarning(FutureWarning): pass

def test():
    warnings.warn("This BETA version will expire on 2018-01-01", BetaWarning)
    print("hello, world!")

if __name__ == '__main__':
    test()
```

```
$ python test.py                          # 默认输出警告信息到 stderr

BetaWarning: This BETA version will expire on 2018-01-01
hello, world!
```

```
$ python -Wi test.py                      # 忽略警告信息
hello, world!
```

```
$ python -We test.py                      # 将警告信息转换为异常

Traceback (most recent call last):
  File "test.py", line 15, in <module>
    test()
  File "test.py", line 11, in test
    warnings.warn("this BETA version will expire on 2018-01-01", BetaWarning)
```

虽然 Warning 继承自 Exception，但默认只输出相关信息，并不会被作为异常抛出。只有开启 -We 参数设置时，Warning 才会被作为异常触发，且可被捕获。

标准库内置了 DeprecationWarning，无须自行定义。不过其默认会被忽略，须-Wd 参数开启显示。

8.2 断言

我们通常会在代码中插入一些防御性（defensive）代码，用来检查数据（参数）是否是指定类型，是否在允许的取值范围内，以避免逻辑发生错误。这对以名字引用，没有类型限制的 Python 似乎更为重要。

只是，在经过严格测试和试运行，确保数据和调用安全后，这些防御性代码基本不再有正面作用，反而会拖累执行性能。如此一来，我们是否需要将其从源码中删除？当然不能！因为代码会随产品持续变更，进入下一个开发和测试迭代周期。最好的方式是通过条件编译，让编译器能自动忽略这些代码。

防御性编程涉及诸多内容，但其根本目标是保护代码本身，而非用户逻辑，所以不适合完成身份验证、权限检查之类的工作。

> 验证用户输入属于业务逻辑，这与面向代码，检查调用方是否按协议传参的防御编程不同。即便是公共类库，也应慎重考虑是否有必要对每次调用参数进行检查，是否要为了万分之一的错误而拖累更多的正确执行。或许，严格的测试覆盖，外加详细的文档和示例会更友好。当然，敏感服务的安全检查则另当别论。

断言概述

多数语言都会提供断言（assert）机制，用于检查所定义的第一条件表达式是否成立。如果失败，则输出或执行第二条件。

```
assert expression1, expression2
```

其实际行为如下：

```
if __debug__:
    if not expression1: raise AssertionError(expression2)
```

通常为每个检查条件单独提供断言语句。这既便于阅读，也有利于调整检查条件，输出有针对性的提示信息。另外，第二表达式作为异常构造参数，可以是简单字符串内容，

或者是其他执行表达式。

```
def int_div(a, b):
    assert isinstance(a, int), "must be an integer"
    assert isinstance(b, int) and bool(b), "must be a non-zero integer"

    return a/b
```

```
>>> int_div(1, 0)
AssertionError: must be a non-zero integer
```

> 除检查参数外，断言有时也用来检查上下文状态。比如，检查数据库连接是否已经建立，以确定执行环境是否符合预期。或替代 print 用作临时测试，即便忘了删除也不会影响执行。

开启优化模式，编译器会直接忽略断言语句，甚至不会为其生成字节码。这远比在运行期进行判断好得多。这有点类似于条件编译机制，仅针对调试版本运行，检查并约束调用行为和数据格式。

```
>>> dis.dis(compile("assert True", "", "exec", optimize = 0))
  1           0 LOAD_CONST               0 (True)
              2 POP_JUMP_IF_TRUE         8
              4 LOAD_GLOBAL              0 (AssertionError)
              6 RAISE_VARARGS            1
        >>    8 LOAD_CONST               1 (None)
             10 RETURN_VALUE
```

```
>>> dis.dis(compile("assert True", "", "exec", optimize = 1))     # 优化模式忽略断言代码
  1           0 LOAD_CONST               0 (None)
              2 RETURN_VALUE
```

> 在命令行下以-O 或-OO 参数启动解释器优化模式。

当然，也正因为断言代码可被忽略，所以其不适合用作固定检查条件，也不适合替代 NotImplementedError 标记那些尚未被实现的算法。从这一点上说，断言和异常用途各异。归根结底，断言面向开发阶段的代码和开发人员，应避免作为运行期逻辑的组成部分。

调试

不仅是断言，实际上任何__debug__逻辑都可能被编译器忽略。

> 默认__debug__=True，开启优化模式后为 False，无法通过赋值修改。

```
code = """
if __debug__:
    print("debug")

print("hello, world")
"""
```

```
>>> dis.dis(compile(code, "", "exec"))
  3           0 LOAD_NAME                0 (print)
              2 LOAD_CONST               0 ('debug')
              4 CALL_FUNCTION            1
              6 POP_TOP

  5           8 LOAD_NAME                0 (print)
             10 LOAD_CONST               1 ('hello, world')
             12 CALL_FUNCTION            1
             14 POP_TOP
             16 LOAD_CONST               2 (None)
             18 RETURN_VALUE
```

```
>>> dis.dis(compile(code, "", "exec", optimize = 1))          # 优化模式忽略调试代码
  5           0 LOAD_NAME                0 (print)
              2 LOAD_CONST               0 ('hello, world')
              4 CALL_FUNCTION            1
              6 POP_TOP
              8 LOAD_CONST               1 (None)
             10 RETURN_VALUE
```

利用这种特性，我们可以编写比断言更复杂的调试版代码。但请注意，无论编译器忽略与否，它都会造成附带影响，比如变量的静态作用域。

test.py

```
def test():
    if __debug__:
        print("debug")
        global x                    # 即便被忽略，依然让代码分析器认为 x 是全局变量
    else:
        print("release")

    x = 100

test()
print(x)

import dis
dis.dis(test)
```

```
$ python test.py                                        # 调试版

debug
100

  6           0 LOAD_GLOBAL              0 (print)
              2 LOAD_CONST               1 ('debug')
              4 CALL_FUNCTION            1
              6 POP_TOP

 11           8 LOAD_CONST               2 (100)
             10 STORE_GLOBAL             1 (x)           # 全局变量
             12 LOAD_CONST               0 (None)
             14 RETURN_VALUE
```

```
$ python -O test.py                                     # 正式版

release
100

  9           0 LOAD_GLOBAL              0 (print)
              2 LOAD_CONST               1 ('release')
              4 CALL_FUNCTION            1
              6 POP_TOP

 11           8 LOAD_CONST               2 (100)
```

```
              10 STORE_GLOBAL             1 (x)              # 全局变量
              12 LOAD_CONST               0 (None)
              14 RETURN_VALUE
```

8.3 上下文

上下文管理协议（Context Management Protocol）是对异常处理结构的一种包装，与分散的处理逻辑相比，它显得更加优雅，也更利于重用。

```
with expression [as var]:
    suite
```

表达式所创建的对象必须包含协议方法__enter__和__exit__，对应的执行流程如下：

伪码

```
o = expression()                        # 1. 创建上下文对象
x = o.__enter__()                       # 2. 执行__enter__，将返回值导出给本地变量 x
                                        #    如果发生异常，则 suite、__exit__不会被执行
try:
    suite                               # 3. 执行用户代码
except:
    typ, val, tb = sys.exc_info()       # 4. 拦截异常，传递给__exit__
finally:
    if not o.__exit__(typ, val, tb):    # 5. 确保__exit__总被执行
        raise val                       #    如果__exit__返回 False，则重新抛出异常
```

设想一下，编写一个数据库上下文对象，需要注意什么？

首先，须在用户逻辑执行前初始化并建立连接；其次，在执行中提供相关操作接口；最后，关闭连接并清理现场。这其中，如何建立连接，如何处理连接池，包括如何清理现场等都与用户逻辑无关。同样，当用户逻辑发生异常时，上下文对象自身如何应对也与用户逻辑无关。

很显然，我们需要将上下文与用户逻辑解耦，让其各自为政。不但用户逻辑无须关心上下文状态；上下文对象也可以不理会用户逻辑是否会抛出异常、是否会捕获异常，我们确保总能适时结束上下文状态即可。

这样一来，上下文状态开启和结束都由其自身负责。无论当前环境有多少上下文对象，都无须担心。这既简化了用户逻辑，又利于在不同场合重用。

```
class DB:
    def __init__(self, name):
        self.name = name

    def __enter__(self):
        print(f"__enter__: {self.name}.open")
        return self

    def __exit__(self, typ, val, tb):
        print(f"__exit__: {self.name}.close; exception: {val}")
        return False


def logic():
    with DB("mysql") as db:
        print(f"exec in {db.name}")
        raise Exception("logic error")
```

```
>>> logic()

__enter__: mysql.open
  exec in mysql
__exit__: mysql.close; exception: logic error
------------------------------------------------------------------------
Exception: logic error
```

可同时使用多个上下文对象。

```
def logic():
    with DB("mysql") as mq, DB("postgres") as pg:
        print(f"exec in {mq.name}")
        print(f"exec in {pg.name}")
        raise Exception("logic error")
```

```
>>> logic()
__enter__: mysql.open
__enter__: postgres.open
  exec in mysql
  exec in postgres
__exit__: postgres.close; exception: logic error
__exit__: mysql.close;    exception: logic error
------------------------------------------------------------------------
Exception: logic error
```

> 如果某个上下文对象__exit__方法返回 True，那么表示它处理了该异常，后续上下文对象将不会接收到该异常信息。请注意多个__exit__按 FILO 顺序执行。

当然，上下文对象未必就要参与用户逻辑，其可仅仅用于测量执行性能，记录日志后重新抛出，或者启动调试器，甚至直接压制异常，等等。

标准库中的很多对象都实现了上下文协议，应优先使用此模式。

```
with open("test.py", "r") as f:           # 确保文件总被关闭
    print(f.read())
```

```
lock = threading.Lock()

for i in range(10):
    with lock:                            # 确保锁总被释放，避免死锁
        print(i)
```

contextlib

上下文对象需要创建类型，并实现协议方法。但已有的代码如何改造？这一点可以借鉴标准库 contextmanager 的做法。

先准备一个示例，其中混杂了上下文和用户逻辑代码。

```
def test(db):
    print(f"open: {db}")                 # 初始化
```

```
    try:
        print(f"exec: {db}")              # 逻辑
    finally:
        print(f"close: {db}")             # 关闭
```

```
>>> test("mysql")
open : mysql
exec : mysql
close: mysql
```

我们要对其进行重构，将上下文剥离出来，以便分离职责并实现重用。

```
@contextlib.contextmanager
def db_context(db):
    try:
        print(f"open: {db}")                # 初始化
        yield db
    finally:
        print(f"close: {db}")               # 关闭


def test(db):
    with db_context(db):
        print(f"exec: {db}")                # 逻辑
        raise Exception("error")            # 模拟用户逻辑异常
```

```
>>> test("mysql")
open : mysql
exec : mysql
close: mysql
---------------------------------
Exception: error
```

从结果来看，我们成功分离了上下文和逻辑代码，实现了最初的设想。而且这里仅仅是拆分函数，并未创建上下文类型，显然起作用的是 contextmanager。下面看看它精简后的示意代码。

contextmanager

```
class GeneratorContextManager:

    def __init__(self, func):
        self.gen = func()                       # 执行原函数（db_context），返回生成器对象

    def __enter__(self):
        return next(self.gen)                   # 启动生成器，返回 yield 结果作为导出变量
                                                # 实际上就是执行 yield 之前的用户代码
    def __exit__(self, typ, val, tb):
        if typ is None:
            next(self.gen)                      # 继续执行 yield 后面的代码
        else:
            self.gen.throw(typ, val, tb)        # 如果有异常发生，则将异常抛回给生成器
```

其设计思路就是利用生成器的分步执行能力，将 db_context 以 yield 为界分割成三部分。

1．yield 之前的代码作为__enter__内容。

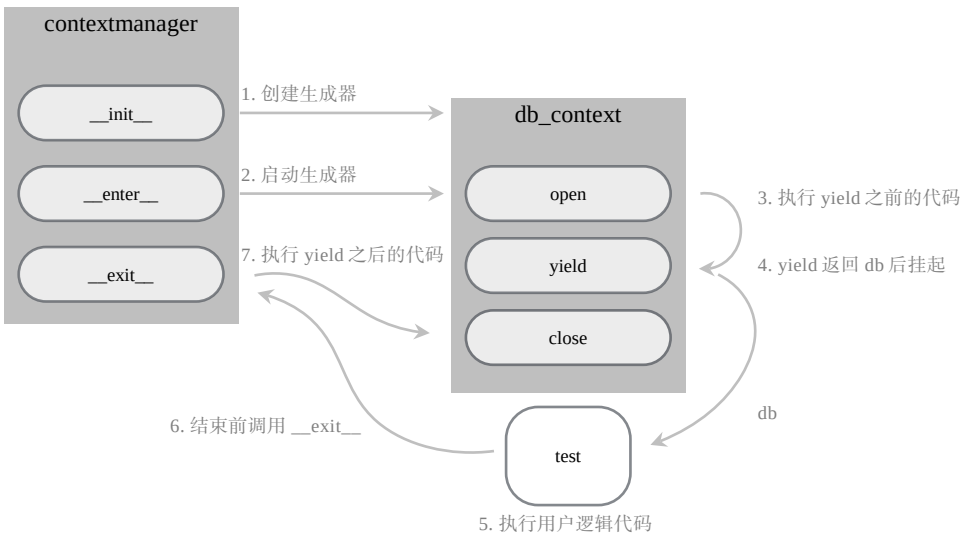
2．yield 返回上下文导出变量。

3．yield 后面的代码作为__exit__内容。

装饰器则创建通用上下文对象，在相关方法中切换执行生成器，完成 3 个步骤的调用。

> 在 test 函数中，with 语句首先调用__enter__方法完成了第 1、2 两个步骤，此时 db_context 生成器因 yield 被挂起，改为执行 test 函数内的用户逻辑代码。直到__exit__被调用，生成器得以重新激活，完成第 3 步。

完整过程的示意图如下：



在 contextlib 模块里，还基于 contextmanager 实现了 closing、suppress 等通用上下文函数。

第 9 章　元编程

元编程将程序当作数据，或在运行期完成编译期的工作。

> Write code that writes code. —— MetaProgramming

9.1　装饰器

对于下面这样一段代码，如果要介入其调用过程，该如何做？

```
def add(x, y):
    return x + y

def test():
    print(add(1, 2))
```

利用装饰器（decorator），可在不侵入内部实现，甚至在不知情的情况下，插入扩展逻辑。

```
@log
def add(x, y):
    return x + y
```

这是因为编译器会将装饰器语法处理成如下模式。

```
def add(x, y):
    return x + y

add = log(add)
```

对按名字搜索执行的调用函数来说，根本不会察觉到被偷天换日。而被调用函数，只要能接收到正确的参数即可。如此一来，我们可在任何时候添加额外的处理过程。从装饰器语法和实现效果来看，这非常类似于 AOP 编程范式。

举例来说，日志、缓存、代理等都不是逻辑必需的功能，其可按职责分离原则封装到不同类型，但如何在没有耦合依赖的情况下进行组装？或者在不修改逻辑和调用的情况下进行调整？除此之外，类似于权限检查、事务管理等是否可从逻辑中剥离，以按需配置形式动态绑定？毕竟，单元测试并不希望有复杂的链式过程。

AOP（Aspect-Oriented Programming，面向切面编程）。就是在不修改目标源码的前提下，添加功能的技术手段或设计模式，它是对 OOP 的补充。其效果就像为相机镜头添加滤镜。

面向对象虽然可通过接口解除对直接类型的依赖，但接口调用依然会出现在代码实现里。这本质上也是一种耦合——功能耦合。试想，在信用卡模块里出现日志调用，怎么说都是突兀且对单元测试不友好的行为。面向切面则不同，它介入的是执行而非实现环节。通过技术手段，拦截调用，提取数据，或者改变执行流程，以低耦合的阻隔方式实现功能变更。从这一点上来说，它按需进行组装和配置，每个都像一片“滤镜”那样插到逻辑和调用之间。

9.1.1 实现

基于装饰器的实际执行方式，可直接以函数实现。

```
def log(fn):
    def wrap(*args, **kwargs):                # 通过包装函数间接调用原函数
        print(f"log: {args}, {kwargs}")
        return fn(*args, **kwargs)

    return wrap                               # 返回包装函数替代原函数与名字关联

@log
def add(x, y):
```

```
    return x + y
```

```
>>> add(1, 2)
log: (1, 2), {}
3
```

其返回包装代理，或仅添加一些额外属性后，原样返回。

```
def log(fn):
    fn.log_func = lambda *args, **kwargs: print(f"log: {args}, {kwargs}")
    return fn
```

同理，任何可调用对象（callable）都可用来实现装饰器模式。

```
class log:
    def __init__(self, fn):
        self.fn = fn

    def __call__(self, *args, **kwargs):
        print(f"log: {args}, {kwargs}")
        return self.fn(*args, **kwargs)

@log
def add(x, y):
    return x + y
```

> 因每条“@log”都会被处理成“log(add)”，也就是每次都会新建一个实例。所以，其应用于多个目标函数完全没问题。

相比于函数，使用类实现似乎可以构建更复杂的装饰器，但这存在一些麻烦。类实现的装饰器应用于实例方法时，会导致方法绑定丢失。

```
class log:
    def __init__(self, fn):
        self.fn = fn

    def __call__(self, *args, **kwargs):
        print(f"log: {args}, {kwargs}")
        return self.fn(*args, **kwargs)
```

```
class X:
    @log
    def test(self): pass
```

```
>>> x = X()
>>> x.test                                  # 方法被装饰器实例所替代
<log at 0x103a05160>

>>> x.test()
TypeError: test() missing 1 required positional argument: 'self'
```

因为装饰器实例替换了方法，结果导致实现绑定的描述符方法被隐藏，无法自动调用。要么让装饰器也实现描述符协议，要么显式传递参数。这样反不如用函数来得简单，因为函数对象默认实现了描述符协议和绑定规则。至于状态维持，函数一样可以添加属性。

```
def log(fn):
    def wrap(*args, **kwargs):
        print(f"log: {args}, {kwargs}")
        return fn(*args, **kwargs)

    return wrap

class X:
    @log
    def test(self): pass
```

```
>>> x = X()
>>> x.test
<bound method log.<locals>.wrap of <X object>>
```

```
>>> log.__get__                                    # 函数默认实现了描述符协议
<method-wrapper '__get__' of function object>
```

嵌套

可对同一目标使用多个装饰器。

```
@a
@b
def test(): pass
```

最终效果是多次嵌套调用。

```
test = a(b(test))
```

装饰器接收前一装饰器的返回值，该返回值可能是包装对象，或是原函数。如此，就须注意排列顺序，因为每个装饰器的返回值并不相同。比如，我们须确保类型方法的装饰器是最外面的一个，因为无法确定内层装饰器如何实现。这可能也会因描述符问题，导致方法绑定失效。

```
class X:

    @classmethod                                # 注意放在最外层
    @log
    def test(cls): pass
```

参数

除被装饰的目标外，还可向装饰器传递其他参数，以实现更多定制特性。

```
@log("demo")
def test(): pass
```

含参数的装饰器，相当于在原装饰器的外面套一个专门用来接收参数的外壳。

```
decorator = log("demo")
test = decorator(test)
```

这样，就多了一次处理过程。

```
def log(name = "default"):                  # 外层函数接收参数
    print(f"args: {name}")

    def decorator(fn):                      # 装饰器
        print(f"decorator: {fn}")
```

```
        return fn                                # 返回包装函数或原函数

    return decorator
```

```
>>> @log("demo")
    def test(): pass

args: demo
decorator: <function test>
```

```
>>> @log()                                       # 对于有参数的装饰器，即便是默认值也需要括号
    def test(): pass

args: default
decorator: <function test>
```

需要注意，@log 和@log( )并不相同。后者依然表示调用外壳函数，无非实参的数量为零。

属性

我们应该让包装函数更像原函数一些，比如拥有某些相同的属性。

```
def log(fn):

    @functools.wraps(fn)                         # 将 fn 的相关属性复制到 wrap
    def wrap(*args, **kwargs):
        return fn(*args, **kwargs)

    print(f"wrap: {id(wrap)}, func: {id(fn)}")
    return wrap


@log
def add(x: int, y: int) -> int:
    return x + y
```

```
>>> add.__name__
add
```

```
>>> add.__annotations__
{'return': int, 'x': int, 'y': int}
```

装饰器 functools.wrap 将原函数__module__、__name__、__doc__、__annotations__等属性复制到包装函数，还用__wrapped__存储原始函数或上一装饰器返回值。可据此判断并绕开装饰器对单元测试的干扰。

```
wrap: 4520685640, func: 4520686456

>>> id(add), id(add.__wrapped__)
(4520685640, 4520686456)
```

类型装饰器

装饰器同样可用于类型，这里的区别无非是接收参数为类型对象而已。

```
def log(cls):

    class wrapper:
        def __init__(self, *args, **kwargs):
            self.__dict__["inst"] = cls(*args, **kwargs)

        def __getattr__(self, name):
            value = getattr(self.inst, name)
            print(f"get: {name} = {value}")
            return value

        def __setattr__(self, name, value):
            print(f"set: {name} = {value}")
            return setattr(self.inst, name, value)

    return wrapper


@log
class X: pass
```

```
>>> x = X()

>>> x.a = 1
```

```
set: a = 1

>>> x.a
get: a = 1
```

> 因为每次都新建 wrapper 类型，所以可应用于不同类型目标。

当然，未必就要返回包装类。这里也可用函数代替，间接调用目标构造方法创建实例。

```
def log(cls):
    def wrap(*args, **kwargs):
        o = cls(*args, **kwargs)
        print(f"log: {o}")
        return o

    return wrap


@log
class X: pass
```

```
>>> X()
log: <__main__.X object at 0x10d7dd208>
     <__main__.X at 0x10d7dd208>
```

9.1.2 应用

利用装饰器功能，我们可以编写各种辅助开发工具，完成诸如调用跟踪、性能测试、内存检测等任务。当然，更多的时候用于模式设计，改善代码结构。

调用跟踪

记录目标调用参数、返回值，以及执行次数和执行时间等信息。

```
def call_count(fn):                          # 调用计数器

    def counter(*args, **kwargs):
        counter.__count__ += 1
```

```
        return fn(*args, **kwargs)

    counter.__count__ = 0
    return counter


@call_count
def a(): pass

@call_count
def b(): pass
```

```
>>> a(); a(); a.__count__
2

>>> b(); b(); b(); b.__count__
3
```

在标准库中有类似的应用，通过缓存结果减少目标执行次数。

```
>>> @functools.lru_cache(10)
    def test(x):
        time.sleep(x)

>>> %%time
    for i in range(1000): test(1)

CPU times: user 668 µs, sys: 1.44 ms, total: 2.1 ms
Wall time: 1 s
```

> 可用装饰器包装 cProfile，像 line_profiler、memory_profiler 那样输出性能测试结果。

属性管理

为目标添加额外属性，在原有设计上以装配方式混入（mixin）其他功能组。

```
def pet(cls):
    cls.dosomething = lambda self: None
    return cls
```

```
@pet                                          # 添加宠物功能
class Parrot: pass
```

> 比起前面章节提及的直接添加基类（__bases__）方式，装饰器显然更优雅一点。
>
> 还可用代理类拦截（__getattr__、__setattr__）对目标进行访问，比如实现只读功能。

实例管理

替代目标构造方法，拦截实例的创建。用于实现对象缓存，或单例模式。

```
def singleton(cls):
    inst = None

    def wrap(*args, **kwargs):
        nonlocal inst
        if not inst: inst = cls(*args, **kwargs)
        return inst

    return wrap


@singleton
class X: pass
```

```
>>> X() is X()
True
```

部件注册

在很多 Web 框架里，用装饰器替代配置文件实现路由注册。

```
class App:
    def __init__(self):
        self.routers = {}

    def route(self, url):                     # 实现为带参数的装饰器，用于注册路由配置
        def register(fn):
            self.routers[url] = fn
```

```
            return fn

        return register
```

```
app = App()

@app.route("/")
def index(): pass

@app.route("/help")
def help(): pass
```

```
>>> app.routers
{'/': <function index>, '/help': <function help>}
```

9.2 描述符

描述符（descriptor）一直被解释器当秘密武器使用。前面提及的属性、方法绑定等内部机制都是描述符在起作用。

不同于实例通用拦截方法（__getattr__等），描述符以单个属性出现，并针对该属性的不同访问行为自动做出响应。最重要的是，描述符能“感知”通过什么引用该属性，从而和目标建立绑定关联。

下面是一个完整描述符实现。

```
class descriptor:

    def __set_name__(self, owner, name):
        print(f"name: {owner.__name__}.{name}")
        self.name = f"__{name}__"

    def __get__(self, instance, owner):
        print(f"get: {instance}, {owner}")
        return getattr(instance, self.name, None)

    def __set__(self, instance, value):
```

```
        print(f"set: {instance}, {value}")
        setattr(instance, self.name, value)

    def __delete__(self, instance):
        print(f"del: {instance}")
        raise AttributeError("delete is disabled")
```

描述符属性必须定义为类型成员，所以其自身不适合存储实例相关的状态。在创建属性时，__set_name__方法被调用，并可通过参数获知目标类型（owner），以及属性名称。

```
>>> class X:
        data = descriptor()

name: X.data
```

以类型或实例访问描述符属性时，__get__被自动调用，且会接收到类型和实例引用。

> 这样就可针对目标进行操作。比如，创建存储字段。

```
>>> x = X()

>>> x.data = 100
set: <X object at 0x1063c6fd0>, 100

>>> x.data
get: <X object at 0x1063c6fd0>, <class 'X'>
100
```

方法__set__、__delete__仅在实例引用时被调用。以类型引用进行赋值或删除操作，会导致描述符属性被替换或删除。

```
>>> X.data
get: None, <class 'X'>

>>> X.data = 100                    # 以类型引用赋值，导致描述符属性被替换
>>> X.data
100
```

还有，将描述符属性赋值给变量或传参时，实际结果是__get__方法的返回值。

```
>>> x = X()

>>> x.data = 100
set: <X object at 0x10627a898>, 100

>>> o = x.data
get: <X object at 0x10627a898>, <class 'X'>

>>> o
100
```

数据描述符

如果定义了__set__或__delete__方法，那么我们便称其为数据描述符（data descriptor）。而仅有__get__的则是非数据描述符（non-data descriptor）。这两者的区别在于，数据描述符属性的优先级高于实例名字空间中的同名成员。

```
class descriptor:

    def __get__(self, instance, owner):
        print("__get__")

    def __set__(self, instance, value):
        print("__set__")


class X:
     data = descriptor()
```

```
>>> x = X()
>>> x.__dict__["data"] = 0

>>> x.data = 100                        # 即便实例的名字空间有同名成员，数据描述符依然优先
__set__

>>> x.data
__get__
```

如果注释掉__set__，使其成为非数据描述符。再执行时，结果就不一样了。

```
>>> x = X()

>>> x.data = 10                          # 同名实例成员的优先级高于非数据描述符
>>> x.data
10

>>> vars(x)
{"data": 10}
```

属性（property）就是数据描述符。就算没有提供 setter 方法，但__set__依然存在，所以其优先级总是高于同名实例成员。

```
>>> p = property()

>>> p.__get__
<method-wrapper '__get__' of property object>

>>> p.__set__
<method-wrapper '__set__' of property object>

>>> p.__delete__
<method-wrapper '__delete__' of property object>
```

方法绑定

因为函数默认实现了描述符协议，所以当以实例或类型访问方法时，__get__首先被调用。

类型和实例作为参数被传入__get__，从而截获绑定目标（__self__），如此就将函数包装成绑定方法对象返回。实际被执行的，就是这个会隐式传入第一参数的包装品。

```
class X:
    def test(self, o): print(o)
```

```
>>> x = X()

>>> x.test
```

```
<bound method X.test of <X object at 0x1063e9ac8>>
```

也就是说方法执行实际分成两个步骤：

```
x.test(123)  --->
        m = x.test.__get__(x, type(x))       # 将函数包装为绑定方法。
        m(123)
              --->  X.test(m.__self__, 123) # 执行时，隐式将 self/cls 参数传给目标函数。
```

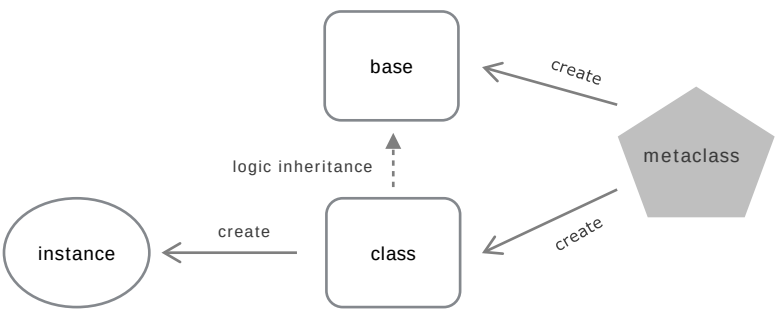
在绑定方法对象内，__self__和__func__存储了执行所需的信息。

```
>>> m = x.test.__get__(x, X)

>>> m
<bound method X.test of <X object at 0x1063e9ac8>>

>>> m.__self__, m.__func__
(<X at 0x1063e9ac8>, <function X.test>)
```

9.3 元类

我们已经知道，类型是实例的创建工厂。那么，又是谁创造了类型对象呢？是父类吗？

确切地说，元类（metaclass）制造了所有的类型对象，并将其与逻辑上的父类关联起来。所以，这里存在两条线：创建和逻辑。



系统默认的元类是 type，所以我们可以描绘出如下过程。

```
base     = type("base", object, ...)
class    = type("class", base, ...)
instance = class(...)
```

```
instance.__class__ is class   and   isinstance(instance, class)
class.__class__    is type    and   isinstance(class, type)
base.__class__     is type    and   isinstance(base, type)
```

> 属性__class__表明该对象由何种类型创建，可用 type(o)返回。

实际上，用 type 直接创建类型对象也无不妥。

```
>>> User = type("User", (object,), {
    "__init__" : lambda self, name: setattr(self, "name", name),
    "test"     : lambda self: print(self.name),
    "table"    : "user",
})

>>> User.__dict__
mappingproxy({'__init__': <function <lambda>>,
              'table': 'user',
              'test': <function <lambda>>,
              ... })
```

```
>>> u = User("yuhen")
>>> u.test()
yuhen

>>> u.__dict__
{'name': 'yuhen'}
```

9.3.1 自定义

可自定义元类，以控制类型对象的生成过程。通常自 type 继承，以 Meta 为后缀名。

```
class DemoMeta(type): pass
```

```
class X(metaclass = DemoMeta): pass            # 使用自定义元类
```

```
>>> X.__class__
DemoMeta
```

Python 3 不再支持 Python 2 __metaclass__语法。

与普通类型构造过程类似，可覆盖构造和初始化方法的定制创建过程。

```
class DemoMeta(type):

    @classmethod
    def __prepare__(cls, name, bases):
        print(f"__prepare__: {name}")
        return {"__make__": "make in DemoMeta"}            # 定制名字空间

    def __new__(cls, name, bases, attrs):
        print(f"__new__: {name}, {bases}, {attrs}")
        return type.__new__(cls, name, bases, attrs)       # 创建并返回类型对象

    def __init__(self, name, bases, attrs):
        print(f"__init__: {self}")
        return type.__init__(self, name, bases, attrs)     # 初始化后，返回类型对象

    def __call__(cls, *args, **kwargs):
        print(f"__call__: {cls}, {args}, {kwargs}")
        return type.__call__(cls, *args, **kwargs)         # 调用类型对象创建实例过程，返回实例
```

按需去实现相关方法。

__prepare__用来创建类型对象名字空间（X.__dict__），可往里添加点什么或使用自定义字典类型。随后，该名字空间会填充其他属性成员，继续传给__new__和__init__方法。

```
class X(metaclass = DemoMeta)   --->
       namespace = DemoMeta.__prepare__(name, ...)
              X = DemoMeta.__new__(name, bases, namespace{attrs})
```

```
                    DemoMeta.__init__(X, ...)
```

__call__ 在类型对象（X）创建其所属的实例时被调用。实际上 X.__new__、X.__init__ 方法就是由此调用的，可用于拦截实例的创建。

```
o = X(1, 2)   --->
        o = DemoMeta.__call__(X, 1, 2)   --->
                o = X.__new__(...)
                    X.__init__(o, ...)
```

```
>>> class X(metaclass = DemoMeta):
        data = 100
        def __init__(self, x, y): pass
        def test(self): pass


__prepare__ : X

__new__     : X, (), {'__make__': 'make in DemoMeta',
                      '__module__': '__main__',
                      '__qualname__': 'X',
                      '__init__': <function X.__init__ at 0x10f815620>,
                      'data': 100,
                      'test': <function X.test at 0x10f815730>}

__init__    : <class '__main__.X'>
```

```
>>> o = X(1, 2)
__call__ : <class '__main__.X'>, (1, 2), {}

>>> o
<__main__.X at 0x10f840a90>
```

当然，这里也可用函数或其他可调用对象代替。

```
def demo_meta(name, bases, attrs):
    print(f"{name}, {bases}, {attrs}")
    return type(name, bases, attrs)
```

```
>>> class X(metaclass = demo_meta):
    data = 100
    def __init__(self, x, y): pass
    def test(self): pass


X, (), {'__module__': '__main__',
        '__qualname__': 'X',
        '__init__': <function X.__init__ at 0x10f8682f0>,
        'data': 100,
        'test': <function X.test at 0x10f868268>}
```

> 函数只是拦截调用，类型对象的创建依然使用 type 完成。但多数时候，这已足够。

参数

这里还可向元类传递参数，实现功能定制。

```
class DemoMeta(type):

    def __new__(meta, name, bases, attrs, **kwargs):
        print(kwargs)
        return type.__new__(meta, name, bases, attrs)
```

```
>>> class X(metaclass = DemoMeta, a = 1, b = "abc"):
        def test(self): pass

{'a': 1, 'b': 'abc'}
```

继承

类型对象的元类设置顺序如下：

1．用 metaclass 显式指定。

2．从基类继承。

3．默认元类 type。

```
class DemoMeta(type): pass

class X(metaclass = DemoMeta): pass
class Y(X): pass                            # 从基类继承元类
```

```
>>> type(Y)
__main__.DemoMeta
```

如果是多继承，则必须保证能继承所有的祖先元类。

```
class AMeta(type): pass
class BMeta(type): pass

class A(metaclass = AMeta): pass
class B(metaclass = BMeta): pass
```

```
>>> class C(A, B): pass

TypeError: metaclass conflict:
    the metaclass of a derived class must be a (non-strict) subclass of
    the metaclasses of all its bases
```

因为__class__只能指向一个元类，所以除非祖先元类之间存在继承关系，否则必然会由于无法访问而导致错误发生。为此，我们需要新建一个元类，让其继承所有的祖先元类。

```
class CMeta(AMeta, BMeta): pass
class C(A, B, metaclass = CMeta): pass
```

9.3.2 应用

基于元类，我们可以实现很多魔法，让对象拥有很高的隐式“智能”，但这会大大提升代码的复杂度。除非有必要，否则不建议这么做。

另外，元类虽然能像普通类型那样为自己的实例提供共享成员，但这里依旧要避免使用。就让元类专注于类型创建和管理，不要掺和逻辑为好。

> Metaclasses are deeper magic that 99% of users should never worry about. If you wonder whether you need them, you don't (the people who actually need them know with certainty that they need them, and don't need an explanation about why).
>
> —— Python Guru Tim Peters on metaclasses.

静态类

阻止类型创建实例对象。

这里直接用__call__拦截实例的创建就行了。

```
class StaticClassMeta(type):
    def __call__(cls, *args, **kwargs):
        raise RuntimeError("can't create object for static class")
```

```
>>> class X(metaclass = StaticClassMeta): pass
RuntimeError: can't create object for static class
```

密封类

阻止类型被继承。

> 将使用该元类，也就是不能被继承的类型添加到集合里，作为后续基类判断条件即可。

```
class SealedClassMeta(type):
```

```
    types = set()

    def __init__(cls, name, bases, attrs):
        if cls.types & set(bases): raise RuntimeError("can't inherit from sealed class")
        cls.types.add(cls)
```

```
>>> class A(metaclass = SealedClassMeta): pass

>>> class B(A): pass
RuntimeError: can't inherit from sealed class
```

9.4 注解

注解（annotation）为函数参数、返回值，以及模块和类型属性添加额外的元数据。

```
def add(x: int, y: int) -> int:
    return x + y
```

```
>>> add.__annotations__
{'return': int, 'x': int, 'y': int}
```

其本质上仅是一种可编程和可执行的注释，在编译期被提取，并与对象相关联。在运行期，其对解释器指令的执行没有任何影响和约束。

```
>>> add("abc", "d")                                    # 并不受注解 int 类型约束
'abcd'
```

```
>>> dis.dis(add)                                       # 字节码指令中也没有任何注解相关的内容
  2           0 LOAD_FAST                0 (x)
              2 LOAD_FAST                1 (y)
              4 BINARY_ADD
              6 RETURN_VALUE


>>> dis.dis(compile("add('abc', 'd')", "", "exec"))    # 和普通函数调用并无二致
  1           0 LOAD_NAME                0 (add)
              2 LOAD_CONST               0 ('abc')
```

```
              4 LOAD_CONST               1 ('d')
              6 CALL_FUNCTION            2
```

注解内容可以是任何对象或表达式。其可应用于变量，但不能用于 lambda 函数。

```
>>> x: int = 123                                        # 注意和等号后的初始化值分开

>>> __annotations__
{'x': int}
```

```
>>> def test(x: {'type': int, 'range':(0, 10)} = 5):   # 等号后为参数的默认值
        pass

>>> test.__annotations__
{'x': {'range': (0, 10), 'type': int}}

>>> test.__defaults__
(5,)
```

```
>>> class X:
        data: int = 10
        def test(self, o: str) -> str: pass


>>> X.__annotations__
{'data': int}

>>> X.test.__annotations__
{'o': str, 'return': str}
```

用途

注解最常见的用途是作为类型和取值范围检查条件，这在 ORM 框架里很常见。

```
def test(x: (int, 0, 10)):
    if __debug__:
        ann_x = test.__annotations__["x"]
        assert isinstance(x, ann_x[0])
        assert ann_x[1] <= x <= ann_x[2]
```

```
    return
```

更优雅的做法是使用装饰器完成，成为一个通用检查器。

其还可以为代码编辑器（PyCharm）和帮助生成工具（pydoc）提供更详细的分类信息。另有 MyPy 等实验型解释器，借助注解实现编译期静态类型检查。

标准库 typing 提供了基于注解的编程支持。

第 10 章 进阶

10.1 解释器

对操作系统、运行时或解释器有所了解是非常必要的。

以 Python 为例，哪怕通过 dis 简单查看解释器的执行过程，也比靠脑补抽象语义，外加 print 输出验证可靠得多。同样，语言或者说某个实现版本总有些不擅长的区域，提前避开去寻找合适的替代方案，要比事后无休止的抱怨有意义，且更节约时间和精力。

当然，这并非要我们深入研究并成长为专家。这就像旅游，提前对目的地的文化、气候以及疾病等有所了解，有助于避开禁忌，在必要时能及时得到救助。

在继续下面的内容以前，你需要下载 CPython 源码。以下示例基于 CPython 3.6.1，内容做了大幅裁减，请以原文件为准。

10.1.1 字节码

字节码被存储在__code__.co_code 里，我们用示例来解释其具体信息。

```
def add(x, y):
    z = x + y
    return z
```

字节码是二进制数据，而非 dis 输出的那种便于阅读的样式。

```
>>> " ".join(f"{b:02X}" for b in add.__code__.co_code)
'7C 00 7C 01 17 00 7D 02 7C 02 53 00'
```

> 指令为单字节，其后有一个参数。

Include/opcode.h

```
#define BINARY_ADD              23                  // 0x17
#define RETURN_VALUE            83                  // 0x53
#define LOAD_FAST              124                  // 0x7C
#define STORE_FAST             125                  // 0x7D
```

这样就可将上面输出的二进制内容分解成字节码指令和参数，并与 dis 输出结果相对应。

> 不同 CPython 版本的指令长度可能不同，CPython 3.6 的每条指令（含参数）为 2 字节。

```
7C 00,  7C 01,  17 00,  7D 02,  7C 02,  53 00
```

```
>>> dis.dis(add)
  2           0 LOAD_FAST                0 (x)         # 7C 00
              2 LOAD_FAST                1 (y)         # 7C 01
              4 BINARY_ADD                             # 17 00
              6 STORE_FAST               2 (z)         # 7D 02

  3           8 LOAD_FAST                2 (z)         # 7C 02
             10 RETURN_VALUE                           # 53 00

行号        偏移量 指令                  参数（目标对象）
```

指令所对应源码的行号存储在两个相关属性中。

```
>>> add.__code__.co_firstlineno          # 起始行号
1

>>> add.__code__.co_lnotab               # 两个一组，分别表示（字节码的偏移位置，相对前一组行号的增量）
```

```
b'\x00\x01\x08\x01'
```

参数通常是对应列表的元素索引，或立即数（整数字面量）。这里可查阅标准库文档的指令说明，其中会提及所使用列表的名称。

```
dis — Disassembler for Python bytecode
   LOAD_FAST(var_num)
   Pushes a reference to the local co_varnames[var_num] onto the stack.

   STORE_FAST(var_num)
   Stores TOS into the local co_varnames[var_num].
```

如此一来，前两条指令参数所指的对象可从下面的列表中获取。

```
>>> add.__code__.co_varnames
('x', 'y', 'z')
```

如果代码中存在循环、跳转，还可以看到 dis 标记出了目标位置。

```
def test():
    for i in range(3): pass
```

```
>>> dis.dis(test)
  7           0 SETUP_LOOP              16 (to 18)    # 参数为相对偏移量，绝对位置要加上自身长度 2
              2 LOAD_GLOBAL              0 (range)
              4 LOAD_CONST               1 (3)
              6 CALL_FUNCTION            1
              8 GET_ITER
        >>   10 FOR_ITER                 4 (to 16)
             12 STORE_FAST               0 (i)
             14 JUMP_ABSOLUTE           10
        >>   16 POP_BLOCK
        >>   18 LOAD_CONST               0 (None)
             20 RETURN_VALUE
```

初步看来，字节码的解读并不麻烦，这比真正的汇编指令要简单许多。这主要是因为，它基于栈式虚拟机设计，没有寄存器等概念。而且，指令转译比较直白，没有另类的优化方式。

字节码并不能被 CPU 执行，实际每条字节码对应一大堆 C 实现的机器指令。不过，我们依然可用栈式虚拟机模型，模拟其执行过程。

CPython 解释器有两个栈：执行栈（Evaluation Stack）存储指令操作数；而块栈（Block Stack）存储循环、异常等信息，以便在不同代码块间快速跳转。

10.1.2 全局锁

全局解释器锁（GIL）是饱受争议的话题。该话题持续时间之长，让人叹为观止。其间各路大神粉墨登场，鼓吹并尝试各种方案。可直到今天，它依旧顽固地存在于最新版的解释器当中，故此不得不说一声“存在即合理”。

全局锁机制让解释器在同一时刻仅有一个线程可被调度执行。对单核环境来说，该实现简单、高效。其易实现对象安全访问，便于进行内存管理和编写扩展。但这同样会导致多核环境下无法实现并行，这对以多线程为基础的并发应用是一个“灾难”。常见的解决方案是用多进程加协程来充分发挥多核环境计算能力。

内置 JIT 的 PyPy 同样有 GIL 存在，不过 PyPy 一直在尝试移除它。

而 Jython、IronPython 得益于平台特性，没有 GIL。

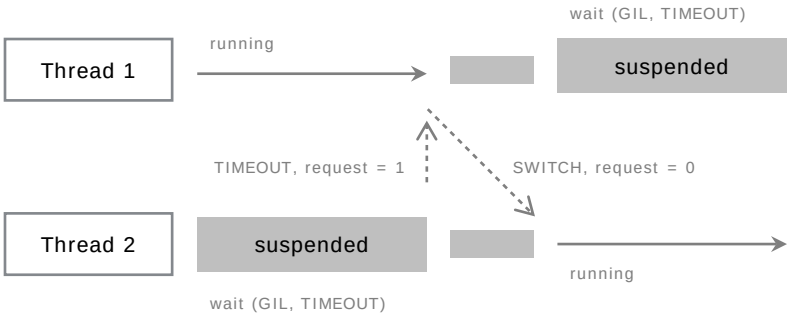
但这并不意味着多线程是 CPython 的禁区。对于 I/O 密集型（I/O-bound）任务，线程会在发生阻塞时主动释放 GIL，以便其他线程得以执行。而对于计算密集型（CPU-bound）任务则采取超时（timeout）策略。

当 GIL 被其他线程占用时，等待线程会阻塞一段时间。如果超时后，依然无法获取锁，则发出请求。这种请求设计得很轻巧，就是一个全局条件变量设置。正在执行的线程在解释循环内会检查该标记，然后释放锁，切换线程执行，其自身进入等待状态。这是很典型的协作机制。

Python/ceval.c

```
PyObject * _PyEval_EvalFrameDefault(PyFrameObject *f, int throwflag)
{
    for (;;) {

        if (_Py_atomic_load_relaxed(&gil_drop_request)) {

            /* Give another thread a chance */
            drop_gil(tstate);

            /* Other threads may run now */

            take_gil(tstate);

            /* Check if we should make a quick exit. */
            if (_Py_Finalizing && _Py_Finalizing != tstate) {
                drop_gil(tstate);
                PyThread_exit_thread();
            }
        }

        switch (opcode) {
            LOAD_FAST ...
            LOAD_CONST ...
        }
    }
}
```

CPython 使用系统线程，且没有实现线程调度。所以，具体哪个等待线程被切换执行，由操作系统决定。甚至，发出请求和被切换执行的也未必就是同一线程。当然，算法保证不会在一个执行周期内多次发出请求。



新 GIL 算法从 CPython 3.2 开始使用，性能较以往有大幅提升。

原来的 tick 机制被 timeout 所替代，其默认超时为 0.005 秒，可使用 sys.setswitchinterval 设置。因为请求检查发生在解释器循环内，所以并不会立即响应，会有一定延时。

如果执行线程调用了 C/C++扩展，那么它可能会阻塞切换。因为机器码指令不像字节码那样使用解释循环，所以也就无法检查请求标记。

虽然 GIL 的存在让多线程无法并行，但依然可实现多任务并发。让我们用两个简单例子，粗略对比一下其执行性能。

不同环境下，测试结果会有一些差异，请以实际运行结果为准。

CPU-bound, test.py

```
import sys
import threading


def countdown():
    n = 100000000
    while n > 0:
        n -= 1


if len(sys.argv) > 1:
    t1 = threading.Thread(target=countdown)
```

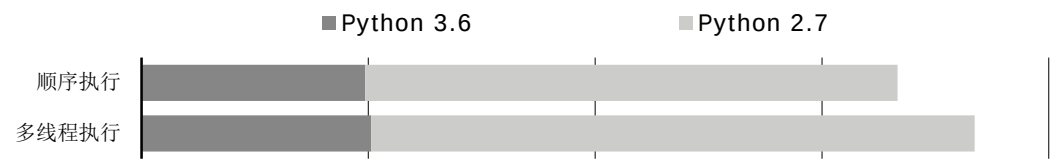
```
        t2 = threading.Thread(target=countdown)

        t1.start()
        t2.start()

        t1.join()
        t2.join()
    else:
        countdown()
        countdown()
```

Python 3.6

```
$ time python3 test.py              # 顺序执行

real    0m14.811s
user    0m14.683s
sys     0m0.046s

$ time python3 test.py 1            # 多线程执行

real    0m15.210s
user    0m15.101s
sys     0m0.130s
```

Python 2.7

```
$ time python2 test.py

real    0m35.205s
user    0m35.023s
sys     0m0.072s

$ time python2 test.py 1

real    0m39.929s
user    0m38.235s
sys     0m11.648s
```



虽然测试环境不太严谨，但依然能看到新版 GIL 的巨大提升。首先，相比于老版本，其总体执行时间大幅缩短；其次，顺序执行和多线程执行之间的差距缩小，表明在锁请求和线程切换处理上的改进效果非常明显。

继续测试 I/O 密集型，看看多线程对性能提升是否有所帮助。

I/O-bound test.py

```
import sys
import threading
import requests               # https://github.com/requests/requests


def test():
    return requests.get("http://cn.bing.com").status_code


if len(sys.argv) > 1:
    ts = [threading.Thread(target=test) for i in range(10)]
    for t in ts: t.start()
    for t in ts: t.join()
else:
    [test() for i in range(10)]
```

```
$ time python3 test.py                # 顺序执行

real    0m2.081s
user    0m0.253s
sys     0m0.079s

$ time python3 test.py 1              # 多线程执行

real    0m0.403s
user    0m0.215s
sys     0m0.046s
```

就算有种种因素影响测试结果，但巨大的时间差依然体现了多线程的优势。所以，要在对的场合使用多线程，而不要因为 GIL 就武断认为多线程绝不能触及。至于计算密集型任务，除使用多进程架构外，还可用 C 来编写多线程扩展，以绕开解释器的限制。

10.1.3 执行过程

基于本书主题和篇幅考虑，这里仅提供一份执行过程的源码简读。

入口

根据命令行参数选择执行模式。

Programs/python.c

```
int main(int argc, char **argv)
{
    Py_Main(argc, argv_copy);
}
```

Modules/main.c

```
int Py_Main(int argc, wchar_t **argv)
{
    // 初始化
    Py_Initialize();

    // 执行模式选择
   if (command) {
        // 命令行模式: -c
        sts = run_command(command, &cf);
    } else if (module) {
        // 模块模式: -m
        sts = (RunModule(module, 1) != 0);
    } else {
        // 交互模式
        if (filename == NULL && stdin_is_interactive) {
            RunStartupFile(&cf);
            RunInteractiveHook();
        }

        // 入口文件模式
```

```
            sts = run_file(fp, filename, &cf);
        }

        // 退出清理
        Py_FinalizeEx();

        return sts;
    }
```

初始化

除内置类型外，最重要的是创建 builtins、sys 模块，并初始化 sys.modules、sys.path 等运行所需的环境配置。

Python/pylifecycle.c

```
    void Py_Initialize(void)
    {
        Py_InitializeEx(1);
    }

    void Py_InitializeEx(int install_sigs)
    {
        _Py_InitializeEx_Private(install_sigs, 1);
    }
```

```
    void _Py_InitializeEx_Private(int install_sigs, int install_importlib)
    {
        // 创建解释器和主线程状态实例
        interp = PyInterpreterState_New();
        tstate = PyThreadState_New(interp);

        // 创建并初始化内置类型
        _Py_ReadyTypes();

        // 初始化带有对象缓存的内置类型
        if (!_PyFrame_Init()) ...
        if (!_PyLong_Init()) ...

        ...

        // 创建 sys.modules，存储运行期被导入的模块
```

```
    interp->modules = PyDict_New();

    // 初始化__builtins__模块
    bimod = _PyBuiltin_Init();
    interp->builtins = PyModule_GetDict(bimod);

    // 初始化内置异常类型
    _PyExc_Init(bimod);

    // 初始化 sys 模块
    sysmod = _PySys_Init();
    interp->sysdict = PyModule_GetDict(sysmod);

    // 设置 sys.path 搜索路径列表（不包括 site-packages）
    PySys_SetPath(Py_GetPath());

    // 设置 sys.modules
    PyDict_SetItemString(interp->sysdict, "modules", interp->modules);

    // 初始化导入机制
    _PyImport_Init();
    import_init(interp, sysmod);

    // 创建__main__模块
    // 注意，此时__main__模块和入口文件并没有什么关系
    initmain(interp);

    // 初始化 site 模块，添加 site-specific 路径到 sys.path 列表
    if (!Py_NoSiteFlag) initsite();
}
```

主模块使用__main__这个特殊名称，但未必就是入口源文件。在交互和命令行模式下，它也作为默认作用域存在。此处的重点是建立名字空间，导入内置成员。

Python/pylifecycle.c

```
static void initmain(PyInterpreterState *interp)
{
    // 创建模块（包括创建__dict__，设置__name__）
    m = PyImport_AddModule("__main__");

    // 将__builtins__添加到名字空间
    d = PyModule_GetDict(m);
```

```
    if (PyDict_GetItemString(d, "__builtins__") == NULL) {
        if (PyDict_SetItemString(d, "__builtins__", bimod) < 0) ...
    }
}
```

执行

在完成初始化后，转入选定执行模式。此处仅列出我们所关心的源文件执行方式。

Modules/main.c

```
static int run_file(FILE *fp, const wchar_t *filename, ...)
{
    run = PyRun_AnyFileExFlags(fp, filename_str, filename != NULL, p_cf);
}
```

Python/pythonrun.c

```
int PyRun_AnyFileExFlags(FILE *fp, const char *filename, ...)
{
    return PyRun_SimpleFileExFlags(fp, filename, closeit, flags);
}
```

使用先前创建的__main__.__dict__作为入口源文件的模块名字空间。

```
int PyRun_SimpleFileExFlags(FILE *fp, const char *filename, ...)
{
    // 获取__main__.__dict__，添加__file__信息
    m = PyImport_AddModule("__main__");
    d = PyModule_GetDict(m);

    if (PyDict_GetItemString(d, "__file__") == NULL) {
        if (PyDict_SetItemString(d, "__file__", f) < 0) ...
    }

    // 运行入口文件（检查 pyc 缓存），将 __main__.__dict__ 作为名字空间传入
    if (maybe_pyc_file(fp, filename, ext, closeit)) {
        v = run_pyc_file(pyc_fp, filename, d, d, flags);
    } else {
        v = PyRun_FileExFlags(fp, filename, Py_file_input, d, d, closeit, flags);
    }
}
```

编译源文件，随后进入字节码执行流程。

```
PyObject * PyRun_FileExFlags(FILE *fp, const char *filename_str, ...)
{
    mod = PyParser_ASTFromFileObject(fp, filename, NULL, ...);
    ret = run_mod(mod, filename, globals, locals, flags, arena);
}

static PyObject * run_mod(mod_ty mod, PyObject *filename, PyObject *globals, ...)
{
    co = PyAST_CompileObject(mod, filename, flags, -1, arena);
    v = PyEval_EvalCode((PyObject*)co, globals, locals);
}
```

创建执行所需的栈帧对象，并准备好参数等执行数据。

Python/ceval.c

```
PyObject * PyEval_EvalCode(PyObject *co, PyObject *globals, PyObject *locals)
{
    return PyEval_EvalCodeEx(co, globals, locals, ...);
}

PyObject * PyEval_EvalCodeEx(PyObject *_co, PyObject *globals, PyObject *locals, ...)
{
    return _PyEval_EvalCodeWithName(_co, globals, locals, ...);
}
```

```
static PyObject * _PyEval_EvalCodeWithName(PyObject *_co, PyObject *globals, ...)
{
    // 创建栈帧对象
    f = PyFrame_New(tstate, co, globals, locals);

    ……填充参数、自由变量（闭包）等数据……

    retval = PyEval_EvalFrameEx(f, 0);
}

PyObject * PyEval_EvalFrameEx(PyFrameObject *f, int throwflag)
{
    //eval_frame: Initialized to PyEval_EvalFrameDefault().
    return tstate->interp->eval_frame(f, throwflag);
}
```

最终，字节码在一个规模接近 3000 行的超大函数内循环解释执行。

> 在这里，我们可以看到任何字节码指令都由很多语句和函数调用组合完成。这也是各方积极寻求 JIT 解决方案的根本原因。毕竟机器码要比字节码快很多，还可使用现成的软硬件优化策略。

Python/ceval.c

```
PyObject * _PyEval_EvalFrameDefault(PyFrameObject *f, int throwflag)
{
    // 指令参数所需的相关名字列表
    co = f->f_code;
    names = co->co_names;
    consts = co->co_consts;
    fastlocals = f->f_localsplus;
    freevars = f->f_localsplus + co->co_nlocals;

    // 类似于 SP、PC 寄存器，下一条指令及帧栈顶位置
    first_instr = (_Py_CODEUNIT *) PyBytes_AS_STRING(co->co_code);
    next_instr = first_instr;
    stack_pointer = f->f_stacktop;

    // 解释循环
    for (;;) {

        // 检查并处理 GIL
        if (_Py_atomic_load_relaxed(&gil_drop_request)) { ... }

    fast_next_opcode:
        // 下一条指令和参数
        NEXTOPARG();

        // 指令执行
        switch (opcode) {
            TARGET(LOAD_FAST) ...
            TARGET(STORE_FAST) ...

            // 每条指令都由 C 实现具体过程
            TARGET(BINARY_ADD) {
                PyObject *right = POP();
                PyObject *left = TOP();
```

```
                PyObject *sum;

                if (PyUnicode_CheckExact(left) && PyUnicode_CheckExact(right)) {
                    sum = unicode_concatenate(left, right, f, next_instr);
                } else {
                    sum = PyNumber_Add(left, right);
                    Py_DECREF(left);
                }

                Py_DECREF(right);
                SET_TOP(sum);

                if (sum == NULL) goto error;
                DISPATCH();                         // goto fast_next_opcode or continue
            }

            ...

        } /* switch */
    } /* main loop */

error:
    ...

exit_eval_frame:
    tstate->frame = f->f_back;
    return _Py_CheckFunctionResult(NULL, retval, "PyEval_EvalFrameEx");
}
```

用户栈内存按地址从低向高分配，每次执行递增指令计数器。

```
#define TOP()             (stack_pointer[-1])
#define SECOND()          (stack_pointer[-2])
#define BASIC_PUSH(v)     (*stack_pointer++ = (v))     // 地址从低向高进行
#define BASIC_POP()       (*--stack_pointer)

#define NEXTOPARG()  do {             \
    _Py_CODEUNIT word = *next_instr;  \
    opcode = _Py_OPCODE(word);        \
    oparg = _Py_OPARG(word);          \
    next_instr++;                     \
} while (0)
```

终止

在结束前，还要完成一系列的清理操作。

Python/pylifecycle.c

```
int Py_FinalizeEx(void)
{
    // 等待前台线程结束
    wait_for_thread_shutdown();

    // 调用 atexit 注册的退出函数
    call_py_exitfuncs();

    // 垃圾回收，执行析构方法
    while (_PyGC_CollectIfEnabled() > 0) /* nothing */;

    // 释放导入的模块
    PyImport_Cleanup();

    // 执行相关结束函数
    _PyTraceMalloc_Fini();
    _PyImport_Fini();
    _PyType_Fini();
    _PyFaulthandler_Fini();

    // 清理解释器和主线程状态
    PyInterpreterState_Clear(interp);

    // 执行内置类型的结束函数
    PyMethod_Fini();
    PyFrame_Fini();
    PyTuple_Fini();
    PyList_Fini();
    PySet_Fini();
    PyDict_Fini();
    PySlice_Fini();

    ...

    return status;
}
```

比如要了解函数调用的相关内容，可先用 dis 反汇编，查看字节码的基本执行过程。然后到解释循环中搜索对应的 CALL_FUNCTION 指令，查证具体实现细节。

阅读 Cpython 源码需要极大的耐心。我们可先制定路线图并圈定范围，以免陷入无休止的跳转当中。

10.1.4 内存分配

在相关源码文件的注释里，已经描述了内存分配器的基本工作原理。

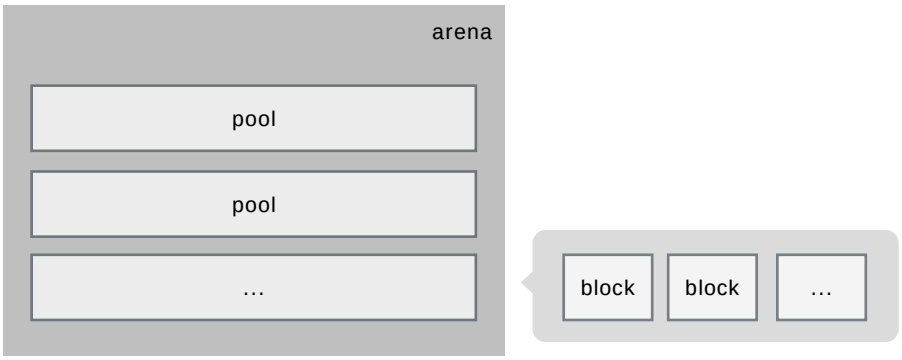
Objects/obmalloc.c

```
   An object allocator for Python.

     Here is an introduction to the layers of the Python memory architecture,
     showing where the object allocator is actually used (layer +2), It is
     called for every object allocation and deallocation (PyObject_New/Del),
     unless the object-specific allocators implement a proprietary allocation
     scheme (ex.: ints use a simple free list). This is also the place where
     the cyclic garbage collector operates selectively on container objects.

     ...
```

以某个阈值为界，分为大小对象。大对象直接分配内存，只有小对象才使用专用内存分配器。小对象按固定长度对齐后，再分成不同类别（size class），以便于复用和管理。

首先，向系统申请大块 Arena 内存，按页大小将其分成多个 Pool 块，这是一级重用单元，每个 Pool 为一种类别的对象提供内存。Pool 被回收后，可重新为另一类别的对象服务。

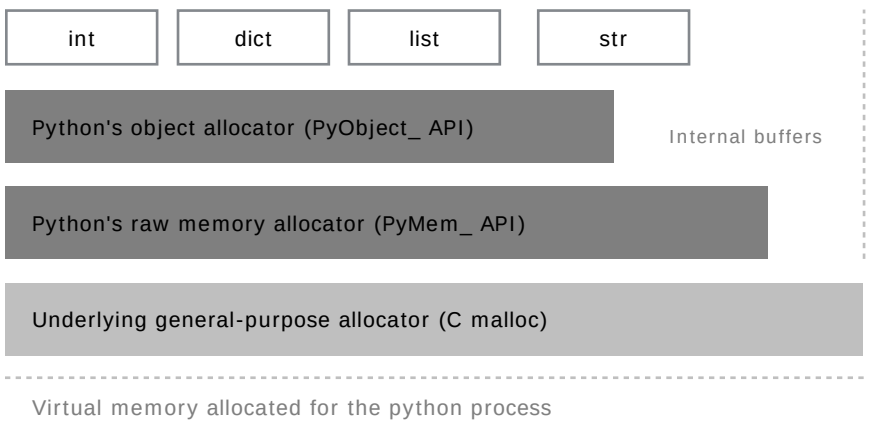
其次，按指定类别大小，将 Pool 再次切分成多个 Block 块。每个 Block 块可存储一个相同类别的小对象，这是二级重用单元。



CPython 3.6：大小对象以 512 字节为界，Arena 为 512 KB，Pool 为 4KB。

如果你阅读过 tcmalloc 等内存分配器源码，那么对此一定很熟悉。简单一点说，就是在通用内存分配基础上，以缓存方式实现小对象快速分配。

在深入细节前，先了解其 API 的基本层次结构，这涉及不同类型的分配选择，也便于我们在源码中查找对应的函数。



小对象按 8 字节对齐，分成 64 个不同类别。

```
#define SMALL_REQUEST_THRESHOLD 512
```

```
   Request in bytes     Size of allocated block      Size class idx
   ----------------------------------------------------------------
         1-8                     8                       0
         9-16                   16                       1
        17-24                   24                       2
        25-32                   32                       3
        33-40                   40                       4
        41-48                   48                       5
        49-56                   56                       6
        57-64                   64                       7
        65-72                   72                       8
         ...                   ...                     ...

      497-504                  504                      62
      505-512                  512                      63

       0, SMALL_REQUEST_THRESHOLD + 1 and up: routed to the underlying allocator.
```

底层 PyMem API 没什么值得关注的内容，让我们直接从 PyObject 开始。

Include/objimpl.h

```
   Functions and macros for modules that implement new object types.

    - PyObject_New(type, typeobj) allocates memory for a new object of the given type,
      and initializes part of it.

    - PyObject_NewVar(type, typeobj, n) is similar but allocates a variable-size
      object with room for n items.

    ...
```

从这些注释的说明信息里，我们很容易找到下一步的目标。

Objects/object.c

```
   PyObject * _PyObject_New(PyTypeObject *tp)
   {
       op = (PyObject *) PyObject_MALLOC(_PyObject_SIZE(tp));
       return PyObject_INIT(op, tp);
   }
```

跳过一堆宏定义后，我们看到了大小对象分界线。

Objects/obmalloc.c

```
static void * _PyObject_Malloc(void *ctx, size_t nbytes)
{
    return _PyObject_Alloc(0, ctx, 1, nbytes);
}
```

```
static void * _PyObject_Alloc(int use_calloc, void *ctx, size_t nelem, size_t elsize)
{
    nbytes = nelem * elsize;
    if (nelem == 0 || elsize == 0) goto redirect;

    if ((nbytes - 1) < SMALL_REQUEST_THRESHOLD) {
        ...
        usable_arenas = new_arena();
        ...
    }

    /* The small block allocator ends here. */

redirect:
    result = PyMem_RawMalloc(nbytes);
    return result;
}
```

暂且放下这些细节。从 Arena 开始，看看最初的大块内存如何申请。

分配流程

每个大块内存都用专门的对象进行管理，以存储其相关状态。

Objects/obmalloc.c

```
static struct arena_object* new_arena(void)
{
    // 没有可用的管理对象……
    if (unused_arena_objects == NULL) {

        // 确定要分配的管理对象数量，并分配内存
        numarenas = maxarenas ? maxarenas << 1 : INITIAL_ARENA_OBJECTS;
        nbytes = numarenas * sizeof(*arenas);
        arenaobj = (struct arena_object *)PyMem_RawRealloc(arenas, nbytes);
```

```
        // 使用链表保存管理对象
        arenas = arenaobj;
        for (i = maxarenas; i < numarenas; ++i) {
            arenas[i].address = 0;
            arenas[i].nextarena = i < numarenas - 1 ?  &arenas[i+1] : NULL;
        }

        unused_arena_objects = &arenas[maxarenas];
    }

    // 提取一个管理对象
    arenaobj = unused_arena_objects;
    unused_arena_objects = arenaobj->nextarena;

    // 为其分配大块内存
    address = _PyObject_Arena.alloc(_PyObject_Arena.ctx, ARENA_SIZE);    // 256KB
    arenaobj->address = (uintptr_t)address;

    // 初始化其 pool 的相关属性
    arenaobj->freepools = NULL;
    arenaobj->pool_address = (block*)arenaobj->address;
    arenaobj->nfreepools = ARENA_SIZE / POOL_SIZE;                       // 4KB
    arenaobj->ntotalpools = arenaobj->nfreepools;

    return arenaobj;
}
```

> 注意：arenaobj 用来管理大块内存，会一次性分配多个备用，以减少内存碎片。但大块内存申请是按请求单次分配的，与管理对象无关。

现在，可以回到小对象内存分配算法了。相比于其他分配器，其架构和实现都非常简单。

优先从对应类别的管理链表中查找正在使用的 Pool。从中提取被回收的 Block 块，或切分未分配的内存。如果失败，则找 Arena 获取新 Pool，然后继续。

```
static void * _PyObject_Alloc(int use_calloc, void *ctx, size_t nelem, size_t elsize)
{
    if ((nbytes - 1) < SMALL_REQUEST_THRESHOLD) {
```

```
    size = (uint)(nbytes - 1) >> ALIGNMENT_SHIFT;

    // 1. 从管理链表（usedpools）提取正在使用的 pool

    pool = usedpools[size + size];
    if (pool != pool->nextpool) {
        ++pool->ref.count;

        // 从回收的复用链表（freeblock）中提取 block
        bp = pool->freeblock;
        if ((pool->freeblock = *(block **)bp) != NULL) {
            return (void *)bp;
        }

        // 从未分配的内存中切分 block
        if (pool->nextoffset <= pool->maxnextoffset) {
            pool->freeblock = (block*)pool + pool->nextoffset;
            pool->nextoffset += INDEX2SIZE(size);
            *(block **)(pool->freeblock) = NULL;
            return (void *)bp;
        }

        ...
    }

    // 2. 从 arean 获取新 pool

    // 申请新的 arena
    if (usable_arenas == NULL) {
        usable_arenas = new_arena();
    }

    // 从回收的复用链表（freepools）中提取 pool
    pool = usable_arenas->freepools;
    if (pool != NULL) {

        // 调整链表和可用计数
        usable_arenas->freepools = pool->nextpool;
        --usable_arenas->nfreepools;

    init_pool:

        // 添加到管理链表（usedpools）
```

```
            next = usedpools[size + size];
            pool->nextpool = next;
            pool->ref.count = 1;

            // 初始化 block 属性
            pool->szidx = size;
            bp = (block *)pool + POOL_OVERHEAD;
            pool->nextoffset = POOL_OVERHEAD + (size << 1);
            pool->maxnextoffset = POOL_SIZE - size;
            pool->freeblock = bp + size;

            // 返回 block
            return (void *)bp;
        }

        // 从 arena 切分新的 pool
        pool = (poolp)usable_arenas->pool_address;
        pool->arenaindex = (uint)(usable_arenas - arenas);
        pool->szidx = DUMMY_SIZE_IDX;

        // 调整位置和数量信息
        usable_arenas->pool_address += POOL_SIZE;
        --usable_arenas->nfreepools;

        // 将 pool 添加到管理链表，并初始化 block 属性
        goto init_pool;
    }

    ...
}
```

释放流程

释放操作只是简单地将 Block 添加到回收链表，但会检查 Pool 和 Arena 是否可被回收，并尝试释放其占用的内存。

Objects/obmalloc.c

```
static void _PyObject_Free(void *ctx, void *p)
{
    // 计算所属的 pool
    pool = POOL_ADDR(p);
```

```
    if (address_in_range(p, pool)) {

        // 将 block 保存到回收的复用链表（freeblock）
        *(block **)p = lastfree = pool->freeblock;
        pool->freeblock = (block *)p;

        if (lastfree) {
            //  检查 pool 被引用的 block 数量
            if (--pool->ref.count != 0) {
                return;
            }

            // 回收自由 pool，归还给 arena 复用链表
            ao = &arenas[pool->arenaindex];
            pool->nextpool = ao->freepools;
            ao->freepools = pool;
            nf = ++ao->nfreepools;

            // 如果 arena 收回全部 pool，则释放其内存
            if (nf == ao->ntotalpools) {
                _PyObject_Arena.free(_PyObject_Arena.ctx, (void *)ao->address, ARENA_SIZE);
                return;
            }

            ...
            return;
        }

        --pool->ref.count;
        return;
    }

redirect:

    // 释放大对象内存
    PyMem_RawFree(p);
}
```

对象缓存

以列表为例，查看其内存分配和回收过程，以及如何缓存复用。

Objects/listobject.c

```
PyObject * PyList_New(Py_ssize_t size)
{
    // 检查复用链表，提取复用对象
    if (numfree) {
        numfree--;
        op = free_list[numfree];
    } else {
        // 没有复用对象，则重新分配
        op = PyObject_GC_New(PyListObject, &PyList_Type);
    }

    // 为元素项单独分配内存
    if (size <= 0)
        op->ob_item = NULL;
    else {
        op->ob_item = (PyObject **) PyMem_Calloc(size, sizeof(PyObject *));
    }

    return (PyObject *) op;
}
```

释放操作，除直接释放元素项内存外，还要检查是否超出复用数量的限制。

```
#define PyList_MAXFREELIST 80
```

```
static void list_dealloc(PyListObject *op)
{
    // 释放元素项内存
    if (op->ob_item != NULL) {
        PyMem_FREE(op->ob_item);
    }

    // 如果未超出限制，则添加到复用列表
    if (numfree < PyList_MAXFREELIST && PyList_CheckExact(op))
        free_list[numfree++] = op;
    else
        Py_TYPE(op)->tp_free((PyObject *)op);
}
```

10.1.5 垃圾回收

下面是解释器内置的两种垃圾回收方式简析。

引用计数

每个对象头部都有一个引用计数器。引用该对象时，计数增加，反之减小。当计数归零时，立即调用回收函数清理并释放内存。此方式实现简单，具备实时响应能力。

Include/object.h

```
typedef struct _object {
    Py_ssize_t          ob_refcnt;
    struct _typeobject  *ob_type;
} PyObject;
```

对计数的操作，以两个宏函数实现。

Include/object.h

```
#define Py_INCREF(op) (                                 \
    ((PyObject *)(op))->ob_refcnt++)
```

```
#define Py_DECREF(op)                                   \
    do {                                                \
        PyObject *_py_decref_tmp = (PyObject *)(op);    \
        if (--(_py_decref_tmp)->ob_refcnt != 0)         \
            _Py_CHECK_REFCNT(_py_decref_tmp)            \
        else                                            \
            _Py_Dealloc(_py_decref_tmp);                \
    } while (0)

#define _Py_Dealloc(op) (                               \
    (*Py_TYPE(op)->tp_dealloc)((PyObject *)(op)))
```

但是，这两个东西非常麻烦。一旦出错或忘记调用，要么计数无法归零导致内存泄漏，要么忘记增加计数让对象提前回收。管理对象生命周期变成了一种挑战。

另外，它还会引发内存管理问题。对象在传递过程中，即便内容未被修改，可引用计数

依然让其内存发生了变动。这可能会导致 CPU 缓存错失率（cache miss）提升，甚至让写时复制（Copy-On-Write）机制失效。

循环引用

为实现循环引用垃圾回收，所有可能导致问题的容器类型，都必须使用专门的内存分配函数。让我们继续以列表类型为例，从创建开始，查看监控和回收过程。

Objects/listobject.c

```
PyObject * PyList_New(Py_ssize_t size)
{
    op = PyObject_GC_New(PyListObject, &PyList_Type);
     _PyObject_GC_TRACK(op);
    return (PyObject *) op;
}
```

Modules/gcmodule.c

```
PyObject * _PyObject_GC_New(PyTypeObject *tp)
{
    PyObject *op = _PyObject_GC_Malloc(_PyObject_SIZE(tp));
    if (op != NULL) op = PyObject_INIT(op, tp);
    return op;
}

PyObject * _PyObject_GC_Malloc(size_t basicsize)
{
    return _PyObject_GC_Alloc(0, basicsize);
}
```

该分配函数会在对象头部附加一个结构，将所有被跟踪对象串成链表。

Include/objimpl.h

```
typedef union _gc_head {
    struct {
        union _gc_head *gc_next;
        union _gc_head *gc_prev;
        Py_ssize_t     gc_refs;
    } gc;
} PyGC_Head;
```

Modules/gcmodule.c

```
_PyObject_GC_Alloc(int use_calloc, size_t basicsize)
{
    PyObject *op;
    PyGC_Head *g;

    // 实际分配内存，包含 GC 头部
    size = sizeof(PyGC_Head) + basicsize;
    if (use_calloc)
        g = (PyGC_Head *)PyObject_Calloc(1, size);
    else
        g = (PyGC_Head *)PyObject_Malloc(size);
    // 初始化 GC 跟踪信息
    g->gc.gc_refs = 0;
    _PyGCHead_SET_REFS(g, GC_UNTRACKED);

    // 检查 G0 是否超过阈值，以确定是否执行垃圾回收
    generations[0].count++;       /* number of allocated GC objects */
    if (generations[0].count > generations[0].threshold && enabled && ...) {
        collect_generations();
    }

    op = FROM_GC(g);
    return op;
}
```

在内存分配完毕后，创建函数还会设置跟踪状态，这里的重点是加入 G0 链表。

Include/objimpl.h

```
/* Tell the GC to track this object. */

#define _PyObject_GC_TRACK(o) do {                  \
    PyGC_Head *g = _Py_AS_GC(o);                    \
    _PyGCHead_SET_REFS(g, _PyGC_REFS_REACHABLE);    \

    g->gc.gc_next = _PyGC_generation0;              \
    g->gc.gc_prev = _PyGC_generation0->gc.gc_prev;  \
    g->gc.gc_prev->gc.gc_next = g;                  \

    _PyGC_generation0->gc.gc_prev = g;              \
} while (0);
```

至此，该对象就处于垃圾回收器的监视之下。当然，如果对象被引用计数回收，则会将其从管理链表中移除。为完成垃圾回收，除为引用计数付费外，还要为可能发生的循环引用付出更大的代价。

Objects/listobject.c

```
static void list_dealloc(PyListObject *op)
{
    PyObject_GC_UnTrack(op);
     Py_TYPE(op)->tp_free((PyObject *)op);
}
```

Modules/gcmodule.c

```
void PyObject_GC_Del(void *op)
{
    PyGC_Head *g = AS_GC(op);
    if (IS_TRACKED(op)) gc_list_remove(g);
    if (generations[0].count > 0) {
        generations[0].count--;
    }
    PyObject_FREE(g);
}
```

在分配函数中，我们看到了阈值检查和回收调用。垃圾回收器将被跟踪对象分成三级代龄，每级都设置了触发阈值和计数器。

Modules/gcmodule.c

```
struct gc_generation {
    PyGC_Head head;
    int threshold;       /* collection threshold */
    int count;           /* count of allocations or collections of younger generations */
};
```

```
#define NUM_GENERATIONS 3
#define GEN_HEAD(n) (&generations[n].head)

static struct gc_generation generations[NUM_GENERATIONS] = {
    /* PyGC_Head,                           threshold, count */
    {{{GEN_HEAD(0), GEN_HEAD(0), 0}},     700,       0},        // G0.t = 分配次数-释放次数
```

```
    {{{GEN_HEAD(1), GEN_HEAD(1), 0}},     10,        0},        // G1.t = G0 回收次数
    {{{GEN_HEAD(2), GEN_HEAD(2), 0}},     10,        0},        // G2.t = G1 回收次数
};
```

即便回收操作是因 G0 触发的，也依然从高级代龄开始检查。

Modules/gcmodule.c

```
static Py_ssize_t collect_generations(void)
{
    // 从 G2 开始检查阈值
    for (i = NUM_GENERATIONS-1; i >= 0; i--) {
        if (generations[i].count > generations[i].threshold) {
            n = collect_with_callback(i);
            break;
        }
    }
    return n;
}

static Py_ssize_t collect_with_callback(int generation)
{
    result = collect(generation, &collected, &uncollectable, 0);
    return result;
}
```

回收并不局限于触发代龄，它会合并下级链表，一次性完成清理操作。通过计算对象引用计数，可将它们分为可达和不可达两类。所有可达（存活）对象转移到高一级代龄，而不可达对象则被释放。随后，参与回收的代龄计数器被归零，后级代龄计数器递增。

如果 G2 阈值被突破，那么将 G0、G1、G2 跟踪列表合并，一次性完成回收操作。

如果只回收 G0，那么 G1.count += 1，因为 G1 的阈值代表 G0 回收执行的次数。

可无论哪级触发，G0 总会参与，其计数器在每次回收后总会被重置。

```
static Py_ssize_t collect(int generation, ...)
{
    // 调整下一级代龄阈值计数
    // 如果此次只回收 G0，那么 G1.count += 1
```

```
    if (generation+1 < NUM_GENERATIONS)
        generations[generation+1].count += 1;

    // 将准备回收代龄阈值的计数归零
    for (i = 0; i <= generation; i++)
        generations[i].count = 0;

    // 将低代龄跟踪对象，全部转移到当前待回收代龄链表
    for (i = 0; i < generation; i++) {
        gc_list_merge(GEN_HEAD(i), GEN_HEAD(generation));
    }

    // 确认存活对象保存到哪一级代龄。如果当前回收 G0，那么保存到 G1
    // 年轻代龄 young 持有本次回收所有需要扫描的对象，old 就是高一级保存存活对象的代龄
    young = GEN_HEAD(generation);
    if (generation < NUM_GENERATIONS-1)
        old = GEN_HEAD(generation+1);
    else
        old = young;

    // 计算可达对象
    update_refs(young);
    subtract_refs(young);

    // 将不可达对象转移到专用列表
    move_unreachable(young, &unreachable);

    // 将可达对象转移到高级代龄
    if (young != old) {
        gc_list_merge(young, old);
    } else {
        untrack_dicts(young);
    }

    ...

    // 回收不可达对象，等等

}
```

这里仅列出了回收的基本过程，相关细节请阅读源码文件。

可通过 gc 相关设置，输出循环引用垃圾回收器的状态，或进行控制。

10.2 扩展

无论是因为性能问题、代码安全，还是为了绕开 GIL 限制，我们总有各种用其他语言编写扩展的理由。

10.2.1 ctypes

如果熟悉 C，那么可以直接编写动态链接库，然后用标准库自带的 ctypes 模块载入调用。且 C 代码中无须引入 Python 类型，无须处理语言和解释器运行特征，其完全与编写普通动态库无异。这是最简单且易维护的一种方式。

注意，不同平台下编译的动态链接库不能跨平台使用。依据相关标准，文件名可能会添加 lib 前缀，并使用特定扩展名（Windows .dll、Linux/BSD .so、macOS .dylib）。

下面编写一个简单的动态库示例，看看如何调用。

demo.h

```
#ifndef __DEMO_H__
#define __DEMO_H__

void hello();

#endif
```

demo.c

```
#include <stdio.h>
#include "demo.h"
```

```
void hello() {
    printf("hello, world!\n");
}
```

```
$ gcc -shared -fPIC -o demo.so demo.c
```

调用前，须先载入动态库。

test.py

```
import ctypes

demo = ctypes.cdll.LoadLibrary("./demo.so")
demo.hello()
```

```
$ python test.py
hello, world!
```

还可直接调用系统运行库 API。考虑到不同平台路径和文件名的差异性，建议通过专用函数查找并返回其完整路径。

```
import ctypes
import ctypes.util

path = ctypes.util.find_library("libc")
libc = ctypes.cdll.LoadLibrary(path)

libc.printf(ctypes.c_char_p(b"hello, world!\n"))
```

> 提示：find_library 搜索路径和平台有关，通常省略 lib 前缀和扩展名。

基本类型

使用一组专用类型，在 C 和 Python 的数据类型间映射和转换。不过鉴于 Python 的实现方式需要更多内存，所以建议减少不必要的转换和调用次数。

```
>>> from ctypes import *
```

```
>>> a = 1234
>>> b = c_int(a)

>>> b
c_int(1234)

>>> b.value, type(b.value)
(1234, int)

>>> b._type_
'i'
```

c_*.value 存储了 Python 原生类型对象，而_type_可用于 array、struct 标准库应用。

数组

使用数组前，须声明其具体类型。数组可以是固定长度，也可动态计算。

```
ArrayType = c_<type> * length
```

demo.c

```
#include <stdio.h>
#include "demo.h"

void each(int nums[], int count) {
    for (int i = 0; i < count; i++) {
        printf("%d\n", nums[i]);
    }
}
```

test.py

```
from ctypes import *

# 定义固定长度的数组类型
Int_Array_3 = c_int * 3

# 创建数组实例，并写入数据
nums = Int_Array_3()
nums[0] = 10
```

```
nums[1] = 20
nums[2] = 30

# C 数组参数是指针类型，须转换一下
demo = cdll.LoadLibrary("demo.so")
demo.each(byref(nums), 3)
```

这里可直接将列表转换为数组。

```
# 数组类型直接以列表长度动态定义
# 展开列表内容作为数组的初始化值，相当于 (c_int * 3)(1, 2, 3)
a = [1, 2, 3]
nums = (c_int * len(a))(*a)

demo = cdll.LoadLibrary("demo.so")
demo.each(byref(nums), 3)
```

或使用 array 类型。

```
from array import array

# 返回内存地址和元素数量后，直接转换为数组指针
nums = array("i", [1, 2, 3])
ptr, count = nums.buffer_info()
nums_p = cast(ptr, POINTER(c_int * count))

# 已经是指针了，无须转换
demo = cdll.LoadLibrary("demo.so")
demo.each(nums_p, count)
```

结构体

要在 Python 里创建与 C 对应的结构类型。

demo.h

```
#ifndef __DEMO_H__
#define __DEMO_H__

typedef struct {
    int x;
    int y;
```

```
} data_t;

void test(data_t *d);

#endif
```

demo.c

```
#include <stdio.h>
#include "demo.h"

void test(data_t *d) {
    d->x = 100;
    d->y = 200;
}
```

从 Structure 继承，使用_fields_定义对应字段。

默认使用与 C 编译器相同的对齐方式，可用_pack_修改。

结构体定义后可作为字段类型嵌入其他数据结构。

---

test.py

```
from ctypes import *

class data_t(Structure):
    _fields_ = [
        ("x", c_int),
        ("y", c_int),
    ]


demo = cdll.LoadLibrary("demo.so")

d = data_t()
demo.test(byref(d))
print(d.x, d.y)
```

指针

用 POINTER 创建指针类型，函数 pointer 返回对象指针。

```
PointerType = POINTER(c_<type>)
```

```
>>> a = c_int(100)
>>> p = pointer(a)

>>> type(p) is POINTER(c_int)
True
```

```
>>> p.contents                          # 相当于 C *p，返回指针的目标对象
c_int(100)

>>> p.contents is p.contents            # 每次构造一个新对象作为返回值
False
```

```
>>> p[0], type(p[0])                    # 返回 Python 原生类型
(100, int)
```

可直接创建指针，然后将其指向目标，并通过它修改原对象。

```
>>> a = c_int(100)

>>> p = POINTER(c_int)()                # 创建指针
>>> p.contents = a                      # 指向目标对象
```

```
>>> p[0] = 200                          # 通过指针修改原对象
>>> a
c_int(200)

>>> p.contents.value = 300              # 直接修改原对象
>>> a
c_int(300)
```

指针索引访问行为与 C 类似。

```
>>> nums = (c_int * 2)()                # 创建数组对象
>>> nums[0] = 100
```

```
>>> nums[1] = 200

>>> p = pointer(nums)                    # 注意是数组指针，而不是元素指针

>>> p[0][0], p[0][1]                     # 先用 p[0]返回数组，然后才是对元素项的索引
(100, 200)

>>> p[0][1] = 20000
>>> nums[1]
20000
```

```
>>> nums = array("i", [1, 2, 3, 4])
>>> ptr, count = nums.buffer_info()

>>> p = cast(ptr, POINTER(c_int))        # 直接转换为 c_int 指针，而不是数组指针
>>> p[2]                                 # 这样就无须先转换为数组了
3
```

相比于 pointer，byref 同样返回目标指针，但其属于轻量级快速实现，用于函数调用。

demo.c

```
#include <stdio.h>
#include <string.h>
#include "demo.h"

void inc(int *x) {
    (*x)++;
}

void str(char *s, size_t count) {
    char *src = "abcdefghijklmnopqrstuvwxyz0123456789";
    memcpy(s, src, count);
}
```

test.py

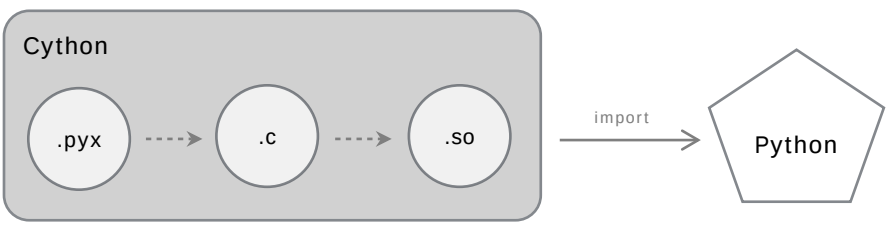
```
from ctypes import *
demo = cdll.LoadLibrary("demo.so")

a = c_int(100)
demo.inc(byref(a))
print(a, a.value)
```

```
s = create_string_buffer(6)
demo.str(s, 6)
print(s.value, str(s.value, "ascii"))
```

10.2.2 Cython

Cython 用 Python 语法编写基于 Python/C API 的标准扩展。

编译器会将 Python 代码转换为 C 代码，然后编译成二进制机器码，这让扩展具备很高的执行性能。另外，还可在代码中直接添加静态类型声明，以绕开 Python 原生对象的额外开销，获得纯 C 的性能优化。也正因为这种转换，还可直接调用 C/C++代码。



Cython 语言是 Python 的超集，所以其理论上可以直接编译原有代码。不过，现阶段这还仅是一个美好的理想，其偶尔会有各种让人头疼的问题。

> 相比于用 C 调用 Python/C API，Cython 无须处理引用计数。

首先，用 pip 进行安装。

```
$ pip install cython
```

然后，写一个示例熟悉一下基本开发过程。

demo.pyx

```
s = "Hello, World!\n"
print(s)
```

setup.py

```
from distutils.core import setup
from Cython.Build import cythonize

setup(
    ext_modules=cythonize("demo.pyx")
)
```

我们从输出信息里可以看到很熟悉的编译命令。

```
$ python setup.py build_ext --inplace

[1/1] Cythonizing demo.pyx
running build_ext
building 'demo' extension

clang -O3 ... -c demo.c -o demo.o
clang ... demo.o -o demo.cpython-36m-darwin.so
```

如果你心情好且有兴趣，可去翻翻转换后的 demo.c 代码。

现在，可以像普通模块那样导入这个二进制扩展了。

```
>>> import demo
Hello, World!

>>> demo.__file__
'.../demo.cpython-36m-darwin.so'
```

自动编译

开发阶段用 setup.py 编译太过麻烦，建议用 pyximport 代替。

先删除上次编译生成的文件，然后安装 pyximport，监测 pyx 模块导入，自动完成编译。

```
>>> import pyximport
>>> pyximport.install()
```

```
>>> import demo
Hello, World!

>>> demo.__file__
'.../.pyxbld/lib.macosx-10.12-x86_64-3.6/demo.cpython-36m-darwin.so'
```

其还可在 IPython、Jupyter Notebook 中直接运行，用来学习和测试都很方便。

```
>>> %load_ext cython
```

```
>>> %%cython
    def add(int x, int y):
        return x + y

>>> add(1, 2)
3

>>> add('a', 'b')
TypeError: an integer is required
```

基本语法

使用 cdef 定义 C 类型变量，或明确声明函数参数的类型。

```
cython
    def test(int x, s):
        cdef int y = x + 5
        print(y, s)
```

```
cython
    cdef int x[3]
    cdef int *p = x

    p[1] = 2
```

相比于默认的 Python 类型，声明为 C 静态类型，有助于提升性能。

编译器会在允许的范围内，自动完成类型转换。

通过不同的关键字，可定义 3 种函数。查看编译转换的 C 代码，可了解其中的差别。

demo.c

```
/*
 * def add(x, y):
 *     return x + y
 */
static PyObject *add(PyObject *self, PyObject *x, PyObject *y)
```

```
/*
 * cdef int c_add(int x, int y):
 *     return x + y
 */
static int c_add(int x, int y)
```

```
/*
 * cpdef int cp_add(int x, int y):
 *     return x + y
 */
static int cp_add(int x, int y, ...)
static PyObject *cp_add(PyObject *self, int x, int y)
```

使用 def 像原生函数，其传入 Python 对象，返回 Python 对象；而 cdef 是纯 C 函数，其可以获得更好的优化、更好的执行性能；剩下的 cpdef 则两者兼顾，其根据参数绑定进行选择。

注意，cdef 定义的变量和函数不会添加到模块的名字空间。

以不同函数定义方式计算斐波那契数列，对比其性能差异。

demo.pyx

```
def fib(n):
```

```
    if n < 2: return n
    return fib(n-2) + fib(n-1)


def fib_int(int n):
    if n < 2: return n
    return fib_int(n-2) + fib_int(n-1)


cdef int _fib_cdef(int n):
    if n < 2: return n
    return _fib_cdef(n-2) + _fib_cdef(n-1)


# 因为 cdef 无法被模块外访问，所以用 def 包装一下
def fib_cdef(int n):
    return _fib_cdef(n)


cpdef fib_cpdef(int n):
    if n < 2: return n
    return fib_cpdef(n-2) + fib_cpdef(n-1)
```

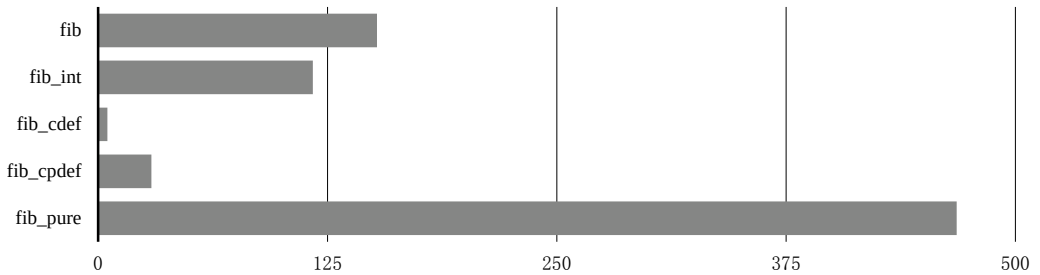
pure.py

```
def fib_pure(n):                                    # 纯 Python 解释实现
    if n < 2: return n
    return fib(n-2) + fib(n-1)
```

先将 demo.pyx 编译成二进制模块，然后使用 timeit 进行测试。

```
$ python -m timeit -s "import demo" "demo.fib(30)"
10 loops, best of 3: 152 msec per loop

$ python -m timeit -s "import demo" "demo.fib_int(30)"
10 loops, best of 3: 117 msec per loop

$ python -m timeit -s "import demo" "demo.fib_cdef(30)"
100 loops, best of 3: 5.29 msec per loop

$ python -m timeit -s "import demo" "demo.fib_cpdef(30)"
10 loops, best of 3: 29 msec per loop
```

```
$ python -m timeit -s "import pure" "pure.fib_pure(30)"
10 loops, best of 3: 468 msec per loop
```



从结果来看，Cython 比 Pure Python 有较大优势。

对于模块内部使用的函数，应尽可能使用 cdef 实现，以获得最佳性能。

调用 C 函数

在 Cython/Include 里已经封装了 C 标准库函数，可直接使用。

cython

```
from libc.stdlib cimport atoi

print(atoi("123"))
```

其还能与 C 混合编程。

hello.h

```
#ifndef __HELLO_H__
#define __HELLO_H__

void hello(char *);

#endif
```

hello.c

```
#include <stdio.h>
```

```
void hello(char *s) {
    printf("hello, %s\n", s);
}
```

demo.pyx

```
cdef extern from "hello.h":
    void hello(char *)

hello("abc")
```

setup.py

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Build import cythonize

setup(ext_modules=cythonize(
    Extension("demo",
              sources=["demo.pyx", "hello.c"],
              include_dirs=["./"]
              )
))
```

编译器会将相关文件全部编译，然后链接到一起。

```
$ python setup.py build_ext --inplace

[1/1] Cythonizing demo.pyx
building 'demo' extension
clang -O3 ... -c demo.c -o demo.o
clang -O3 ... -c hello.c -o hello.o
clang ... demo.o hello.o -o demo.cpython-36m-darwin.so
```

```
$ python -c "import demo"
hello, abc
```

> 当然，也可以将 hello.c 编译成动态链接库给 demo.pyx 使用。

Cython 虽然使用 Python 语法，但其不是零成本，依旧需要学习。且当前版本尚有许多

问题需要解决，这可能会影响其开发进度，甚至会出现无法解决的缺陷。建议读者先在个人项目或者规模较小的算法内使用。如果只是局部性能问题，还可试试 Numba JIT 编译。

Numba gives you the power to speed up your applications with high performance functions written directly in Python. With a few annotations, array-oriented and math-heavy Python code can be just-in-time compiled to native machine instructions, similar in performance to C, C++ and Fortran, without having to switch languages or Python interpreters.

Numba works by generating optimized machine code using the LLVM compiler infrastructure at import time, runtime, or statically (using the included pycc tool). Numba supports compilation of Python to run on either CPU or GPU hardware, and is designed to integrate with the Python scientific software stack.

The Numba project is supported by Anaconda, Inc.

第 11 章 测试

11.1 单元测试

单元测试（unit testing）用来进行正确性检测，以确保代码和算法符合预期设计。

单元测试基于最小单元，也就是程序的基本部件，比如函数、类等。在隔离测试各部件，确保底层正确的前提下，再行组合向上完成套件测试。这样可及早发现问题，及时进行更新和调整。

测试持续跟踪代码的变更，报告和定位错误。我们应使用全面的测试数据，覆盖算法的全部路径。从某种程度上说，测试是面向开发人员，对产品设计的另一种描述。同时，测试也是程序服务的使用说明。因此，测试应与产品设计同步，且具备良好的可读性。

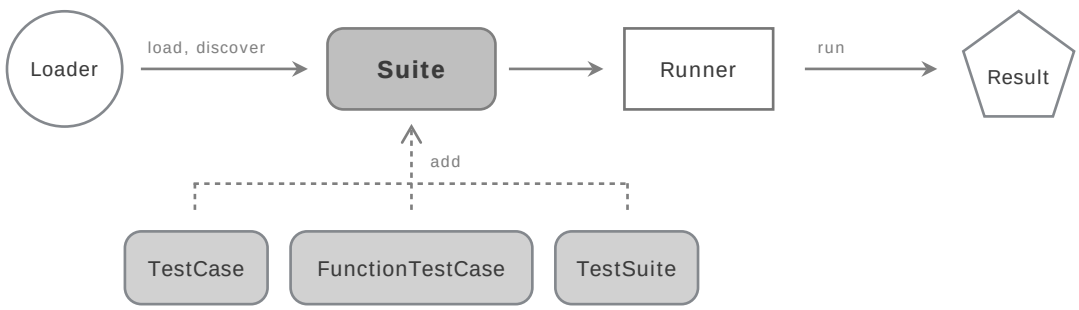
测试驱动开发（TDD）要求在编写代码之前，先构建测试，但这并不容易。多数时候，需求和架构的不成熟会导致算法剧烈变化，这样很难先行将测试确定下来。可以考虑和编码同步，在重构过程中剥离出测试用例。两厢对照，还可要求代码必须具备可测试性。

请注意，测试用例要检查的是错误，而非目标正确返回。另外，单个用例要足够简单，仅测试一项功能，否则其本身就难以维护。可将复杂功能分解，以测试套件的方式进行

打包。另外，还应避免测试目标的私有成员，这会导致过高的耦合，让代码重构变得困难。同样，第三方库也不是测试目标，或许应该有一个基于接口的隔离代理。

11.1.1 unittest

整个框架由多种不同用途的组件组成。其可直接在命令行中使用，也可编码定制自动测试流程。



通常将测试放在独立目录（test 或 tests）下，文件名包含特定前后缀（test_*.py, *_test.py）。

- TestCase：测试用例（实现一到多个测试方法）。
- TestSuite：测试套件（组合多个用例或套件）。
- TestLoader：加载器（查找测试方法）。
- TestRunner：执行器（执行测试，返回测试结果）。
- TestResult：测试结果。

测试用例

其须直接或间接继承自 TestCase、FunctionTestCase，并按惯例添加 Test 或 TestCase 后缀。

方法 run 仅调用构造参数（methodName = 'runTest'）指定的测试方法。

加载器可以按特定规则查找所有测试方法，分别为每个方法创建单独的实例。

demo_test.py

```
import unittest

class DemoTest(unittest.TestCase):

    def runTest(self):
        self.assertTrue(True)

    def test_1(self):
        self.assertTrue(True)

    def test_2(self):
        self.assertFalse(True)


print(DemoTest().run())                    # 测试 runTest
print(DemoTest("test_1").run())            # 测试 test_1
```

```
$ python demo_test.py

<unittest.result.TestResult run=1 errors=0 failures=0>
<unittest.result.TestResult run=1 errors=0 failures=0>
```

除断言方法外，还有一些特殊的装饰器和上下文。

```
class DemoTest(unittest.TestCase):

    @unittest.expectedFailure                  # 期望测试失败
    def test_1(self):
        self.assertTrue(False)

    @unittest.skip("reason...")                # 忽略该测试（另有几个可指定条件的版本）
    def test_2(self):
        self.assertTrue(False)

    def test_3(self):                          # 测试是否抛出指定的异常
        with self.assertRaises(Exception):
            raise Exception()
```

```
$ python -m unittest -v demo_test.py
```

```
test_1 (demo_test.DemoTest) ... expected failure
test_2 (demo_test.DemoTest) ... skipped 'reason...'
test_3 (demo_test.DemoTest) ... ok
----------------------------------------------------
OK (skipped=1, expected failures=1)
```

初始化和清理

有几个被称作 Fixture 的特殊方法，可为测试提供环境初始化和结束清理操作。

```
class DemoTest(unittest.TestCase):

    @classmethod
    def setUpClass(cls):
        print(">>> class setup")

    @classmethod
    def tearDownClass(cls):
        print(">>> class teardown")

    def setUp(self):
        print("[setup]")

    def tearDown(self):
        print("[teardown]")

    def test_1(self):
        print("hello, test_1!")

    def test_2(self):
        print("hello, test_2!")
```

```
$ python -m unittest -v demo_test.py

>>> class setup

test_1 (demo_test.DemoTest) ...
[setup]
hello, test_1!
[teardown]
ok
```

```
test_2 (demo_test.DemoTest) ...
[setup]
hello, test_2!
[teardown]
ok

>>> class teardown
```

每个测试方法执行前后，都会调用 setUp/tearDown。

但无论有多少个实例，setUpClass/tearDownClass 仅执行一次。

还可为模块添加 setUpModule/tearDownModule。

如果 Fixture 方法引发异常，会影响其对应级别的测试。

函数用例

如果有现成的测试函数，则可直接以 FunctionTestCase 创建实例包装。

FunctionTestCase 继承自 TestCase，其是专门为函数准备的通用测试类型。

将函数作为参数创建测试实例，加入套件，或直接运行。

```
def testSomething():
    assert True

result = unittest.FunctionTestCase(testSomething).run()
print(result)
```

子测试

默认情况下，测试方法在第一次断言失败时终止。但某些时候，我们须测试一组数据，汇总所有引起错误的原因。这就要求测试方法必须忽略失败，完成全部数据后才能结束。

使用 subTest，将失败控制在子测试内，这样就可以遍历整个数据组了。

```
class DemoTest(unittest.TestCase):

    def test_subtest(self):
        data = range(5)
        for i in data:
            with self.subTest(i=i):                    # 子测试（参数用于输出）
                self.assertEqual(i % 2, 0)
```

```
$ python -m unittest demo_test.DemoTest.test_subtest     # 完成全部测试后进行汇总

FAIL: test_subtest (demo_test.DemoTest) (i=1)
FAIL: test_subtest (demo_test.DemoTest) (i=3)
----------------------------------------------
FAILED (failures=2)
```

测试套件

将多个用例或套件的实例组合起来，完成产品功能组级别的批量测试。

TestSuite.run 循环调用所有测试组件的 run 方法。

TestCase、TestSuite 实例是可调用对象，用__call__调用 run 方法。

按设计要求，分别为每个参与的测试方法创建实例，并加入套件。

```
class UserTest(unittest.TestCase):

    def test_register(self):
        self.assertTrue(True)

    def test_login(self):
        self.assertTrue(True)


class CartTest(unittest.TestCase):
```

```
    def test_add(self):
        self.assertTrue(True)


class OrderTest(unittest.TestCase):

    def runTest(self):
        self.assertTrue(True)


suite = unittest.TestSuite()
suite.addTest(UserTest("test_login"))
suite.addTest(CartTest("test_add"))
suite.addTest(OrderTest())

unittest.TextTestRunner(verbosity=2).run(suite)
```

```
$ python demo_test.py

test_login (__main__.UserTest) ... ok
test_add (__main__.CartTest)   ... ok
runTest (__main__.OrderTest)   ... ok
----------------------------------------
OK
```

如果只是批量测试单个用例的方法，那么让其自己管理更合适一些。

创建单独的方法，或覆盖 runTest，然后用子测试完成。

```
class DemoTest(unittest.TestCase):

    def test_1(self):
        self.assertTrue(True)

    def test_2(self):
        self.assertTrue(True)

    def runTest(self):
        tests = (self.test_1, self.test_2)
        for t in tests:
```

```
            with self.subTest(test=t): t()          # 使用子测试，确保所有测试方法都会被执行
```

加载器

加载器（TestLoader）有多种发现机制，查找测试方法，为其创建用例实例，并组合成套件返回。使用命令行时，实际就是由加载器完成相关工作。

其可递归扫描目录，查找符合匹配条件的测试模块；或指定具体的模块、用例类型，甚至是某个测试方法。

完整的流程是用 discover 递归目录，查找所有文件名相符的模块。随后用 loadTestsFromModule 在模块内获取所有用例类型，再以 loadTestsFromTestCase 为用例的全部测试方法创建实例。最终，将这些组合成测试套件交给执行器。

loadTestsFromTestCase 调用 getTestCaseNames 查找类型中包含特定前缀（testMethodPrefix）的测试方法，没找到时才选择 runTest。

loadTestsFromModule 按照加载协议（Load Tests Protocol）约定，先调用 load_tests 函数返回自定义测试套件。仅在没有协议实现时，才返回模块内的所有用例类型。

可创建加载器对象，或直接使用默认的 defaultTestLoader 实例。

```
class ATest(unittest.TestCase):

    def runTest(self):
        self.assertTrue(True)

    def test_1(self):
        self.assertTrue(True)

    def test_2(self):
        self.assertFalse(True)


class BTest(unittest.TestCase):

    def runTest(self):
        self.assertTrue(True)
```

```
loader = unittest.defaultTestLoader
print("Case:", loader.loadTestsFromTestCase(ATest))
print("Module:", loader.loadTestsFromModule(sys.modules[__name__]))
```

```
$ python demo_test.py

Case: <TestSuite tests=[
    <ATest testMethod=test_1>,
    <ATest testMethod=test_2>
]>

Module: <TestSuite tests=[
    <TestSuite tests=[
        <ATest testMethod=test_1>,
        <ATest testMethod=test_2>
    ]>,
    <TestSuite tests=[
        <BTest testMethod=runTest>
    ]>
]>
```

加载协议

在包或模块内实现 load_tests 函数，返回定制的测试套件。

---

load_tests (loader, standard_tests, pattern)

standard_tests：以默认方式从该模块找到的测试方法套件，可直接返回或扩展。

pattern：默认为 discover 文件名匹配模式，也可在 loadTestsFromModule 调用时设定。

```
def load_tests(loader, standard_tests, pattern):
    suite = unittest.TestSuite()
    suite.addTests(map(ATest, ("test_1", "test_2")))
    return suite


loader = unittest.defaultTestLoader
```

```
print(loader.loadTestsFromModule(sys.modules[__name__]))
```

```
$ python demo_test.py                    # 返回套件内并未包含 BTest

<TestSuite tests=[
    <ATest testMethod=test_1>,
    <ATest testMethod=test_2>
]>
```

执行器

执行器（TestRunner）接收用例或套件，执行测试并返回结果（TestResult）。

> TextTestRunner 默认输出到 sys.stderr，可使用 stream 参数将结果保存到文件中。

```
class DemoTest(unittest.TestCase):

    def test_1(self):
        self.assertTrue(True)

    def test_2(self):
        self.assertFalse(True)


loader = unittest.defaultTestLoader
suite = loader.loadTestsFromTestCase(DemoTest)
unittest.TextTestRunner(verbosity = 2).run(suite)
```

```
$ python demo_test.py

test_1 (__main__.DemoTest) ... ok
test_2 (__main__.DemoTest) ... FAIL
----------------------------------
FAILED (failures=1)
```

命令行

通常直接在命令行执行测试，也可集成到自动化工具。

```
python -m unittest test_module

python -m unittest module.TestClass

python -m unittest module.Class.test_method

python -m unittest path/to/test_file.py

python -m unittest discover test_path "*_test.py"
```

如有定制需要，可在测试模块内执行 unittest.main，其方式与命令行类似。

```
if __name__ == '__main__':
    unittest.main(verbosity = 2)
```

11.1.2 unittest.mock

如果测试目标依赖其他对象或模块，我们就可能需要以模拟（mock）替代它。

理由如下：

- 依赖对象未完成，也许仅有接口。
- 依赖数据库、网络等复杂度较高的部署场景。
- 依赖非固定测试场景，比如数据不恒定。
- 依赖服务的错误很难触发，比如某种网络故障。
- 将依赖隔离，确保问题是由测试目标引发的。

某些时候，甚至要以模拟对象替代测试目标。比如先行完成测试逻辑，解除开发次序依赖。或在编写测试逻辑的过程中，用模拟对象来完善目标接口设计。

下面先看一个简单示例。我们须用模拟对象替代真实网络请求。这是因为其速度和返回值依赖具体的外部网络环境，无法固定，且不能保证错误是由测试目标引发的。这些问题给测试任务带来了不确定因素。

```
import requests

def logic(url):
    resp = requests.get(url)
    return resp.status_code
```

用 mock 替换 requests.get，直接返回固定结果 status_code。

使用 Fixture 保存被替换的原始对象，以便将来恢复。

```
from types import SimpleNamespace
from unittest import TestCase
from unittest.mock import Mock


class DemoTest(TestCase):

    def setUp(self):
        # 保存被模拟替换的原对象，以便将来恢复
        self._request_get = requests.get

    def tearDown(self):
        requests.get = self._request_get

    def test_logic(self):
        # 固定测试数据
        data = SimpleNamespace(
            url  = "http://www.abc.com",
            code = 200
        )

        # 创建模拟对象，设置有 status_code 属性的返回值
        get = Mock()
        get.return_value.status_code = data.code

        # 用模拟对象替换网络请求方法
        requests.get = get

        # 测试逻辑
        self.assertEqual(logic(data.url), data.code)
```

```
$ python -m unittest -v demo_test.DemoTest.test_logic

test_logic (demo_test.DemoTest) ... ok
```

Mock

Mock 以__getattr__拦截被替换对象的属性访问，动态创建所需成员。且新建成员同是模拟类型，以实现链式属性设置和访问。

```
>>> m = Mock()

>>> m
<Mock id='4523783952'>

>>> m.name
<Mock name='mock.name' id='4521063872'>
```

> 默认不能以 assert 作为属性名字的前缀，这可用 unsafe 参数控制。

```
>>> m.assertX
AttributeError: assertX

>>> m = Mock(unsafe = True)

>>> m.assertX
<Mock name='mock.assertX' id='4522751424'>
```

return_value

通过设置 return_value 属性，使其成为可调用对象，用来模拟函数和方法。

> 其内部还会记录调用信息，支持断言检查，包括调用参数和次数等。

```
>>> m = Mock()
>>> m.test.return_value = 100

>>> m.test(1, 2, 3)
```

```
100
```

```
>>> m.test.assert_called_with(1, 2)
AssertionError:
    Expected call: test(1, 2)
    Actual call: test(1, 2, 3)

>>> m.test.assert_called_with(1, 2, 3)
```

如果是普通字段，则直接赋值即可。

```
>>> m = Mock()
>>> m.name = "user1"

>>> m.name
'user1'
```

side_effect

构造参数 side_effect 指定可调用对象、迭代器或异常，用来替代 return_value 返回。

```
>>> m = Mock(side_effect = lambda a: a + 1)
>>> m(10)
11
```

```
>>> m = Mock()
>>> m.next = Mock(side_effect = [1, 2, 3])

>>> m.next()
1

>>> m.next()
2
```

```
>>> m = Mock()
>>> m.test = Mock(side_effect = KeyError("xxx"))

>>> m.test()
KeyError: 'xxx'
```

如果可调用对象返回 unittest.mock.DEFAULT，那么实际返回 return_value。

将 side_effect 设置为 None，其影响消失，则总是返回 return_value。

```
>>> m = Mock(side_effect = lambda x: 100 if x > 0 else DEFAULT, return_value = -1)

>>> m(1)
100

>>> m(0)
-1
```

```
>>> m.side_effect = None

>>> m(1)
-1
```

spec

可从列表或某个类型里提取可用属性名字的清单，用以约束模拟对象。

```
>>> m = Mock(spec = ["name", "test"])

>>> m.name
<Mock name='mock.name' id='4521063760'>

>>> m.test
<Mock name='mock.test' id='4524246800'>

>>> m.abc
AttributeError: Mock object has no attribute 'abc'
```

```
>>> class A:
        name = "a"
        def hello(self): pass

>>> m = Mock(spec = A)

>>> m.name
<Mock name='mock.name' id='4524247416'>
```

```
>>> m.hello
<Mock name='mock.hello' id='4522750640'>

>>> m.abc
AttributeError: Mock object has no attribute 'abc'
```

但是，spec 并不能阻止通过赋值创建属性。

```
>>> m = Mock(spec = ["name"])
>>> m.abc
AttributeError: Mock object has no attribute 'abc'

>>> m.abc = 1
>>> m.abc
1
```

可改用 spec_set 阻止这种行为，实现严格约束。

```
>>> m = Mock(spec_set = ["name"])

>>> m.name
<Mock name='mock.name' id='4524652640'>

>>> m.abc = 1
AttributeError: Mock object has no attribute 'abc'
```

对函数和方法而言，还要约束参数列表，使其和目标设计保持一致。

```
>>> class A:
        def test(self, a, b): return 1

>>> m = create_autospec(A, spec_set = True, instance = True)
>>> m.test.return_value = 1

>>> m.test(1)
TypeError: missing a required argument: 'b'

>>> m.test(1, 2)
1
```

```
>>> m = Mock()
>>> m.test = create_autospec(lambda a, b: None , return_value = 100)

>>> m.test(1)
TypeError: missing a required argument: 'b'

>>> m.test(1, 2)
100
```

wraps

通过 wraps 参数，可以将模拟访问传递给真实对象。这样可使用开关在模拟和真实对象间切换，而非删除模拟代码。

```
>>> class A:
        def add(self, a, b):
            return a + b

>>> m1 = Mock(spec_set = A, wraps = A())
>>> m1.add(10, 20)
30
```

可一旦设置 return_value，则不再传递给真实对象，而是直接返回。

```
>>> m1.add.return_value = 1
>>> m1.add(1, 2)
1
```

MagicMock

额外提供一个 mock_add_spec 方法，用于按需调整 spec 设置。

```
>>> m = MagicMock()

>>> m.name
<MagicMock name='mock.name' id='4524436952'>
```

```
>>> m.mock_add_spec(["test"], True)                    # 以 spec_set 方式设置（不是新增）

>>> m.test
```

```
<MagicMock name='mock.test' id='4524433536'>

>>> m.name
AttributeError: Mock object has no attribute 'name'

>>> m.name = "abc"
AttributeError: Mock object has no attribute 'name'
```

```
>>> m.mock_add_spec(["name", "test"], True)

>>> m.name
<MagicMock name='mock.name' id='4524436952'>

>>> m.test
<MagicMock name='mock.test' id='4524433536'>
```

```
>>> m.mock_add_spec([], True)                          # 阻止所有属性访问

>>> m.name
AttributeError: Mock object has no attribute 'name'
```

```
>>> m.mock_add_spec(None)                              # 取消 spec 设置

>>> m.name
<MagicMock name='mock.name' id='4524436952'>

>>> m.test
<MagicMock name='mock.test' id='4524433536'>
```

Patch

用模拟对象替换真实对象后，须及时恢复，以免影响后续逻辑。

> 相比于 Fixture，自带的 patch 上下文更合理一些，可将替换和恢复约束在更小的作用域内。

让我们用 patch 改写本节开始的那个示例。

```
from types import SimpleNamespace
from unittest import TestCase
```

```
from unittest.mock import Mock, patch


class DemoTest(TestCase):

    def test_logic(self):
        data = SimpleNamespace(url = "http://www.abc.com", code = 200)

        with patch("requests.get") as get:
            get.return_value.status_code = data.code
            self.assertEqual(logic(data.url), data.code)
```

这里默认创建一个 MagicMock 对象，作为__enter__返回值。也可用其他对象替代，比如函数。

```
class DemoTest(TestCase):

    def test_logic(self):
        data = SimpleNamespace(url = "http://www.abc.com", code = 200)

        with patch("demo_test.logic", lambda url: data.code) as m:
            self.assertEqual(m(data.url), data.code)
```

其还可作为装饰器使用。

默认返回 MagicMock 对象，作为测试方法参数传入。

如果设置了 new 参数，则直接替换，而不会有新参数传入测试方法。

```
class DemoTest(TestCase):

    @patch("requests.get")
    def test_logic(self, get):
        data = SimpleNamespace(url = "http://www.abc.com", code = 200)

        get.return_value.status_code = data.code
        self.assertEqual(logic(data.url), data.code)
```

```
class DemoTest(TestCase):
```

```
    @patch("demo_test.logic", lambda url: 200)
    def test_logic(self):
        data = SimpleNamespace(url = "http://www.abc.com", code = 200)
        self.assertEqual(logic(data.url), data.code)
```

11.1.3 coverage

标准库没有代码覆盖率（code coverage）测试工具，因此须借助第三方扩展。

Coverage.py https://github.com/nedbat/coveragepy

其使用方法很简单，可直接在命令行执行。

```
$ coverage run --source . -m unittest demo_test.DemoTest      # 仅测试当前目录下的文件

$ coverage report                                             # 以文本方式查看结果报告
$ coverage html                                               # 生成 HTML 文件
```

11.2 性能测试

11.2.1 timeit

timeit 为标准库自带，用于测量少量代码片段的执行时间。其基本算法思路就是执行足够多的次数，从中选取最小平均值，以缩小测量误差。

算法如下：

1．将代码循环执行多次（number），以便有足够长的统计时间。

2．将步骤 1 循环执行多次（repeat），获取足够多的统计采样。

3．从步骤 2 的结果选取耗时最短的采样，计算单次平均值（min_time / number）。

> 目标代码被插入测量函数模板进行预编译，垃圾回收器临时关闭，初始化代码（setup）在测量计时器开启前执行，等等，尽一切可能排除干扰因素。在输出结果前，还会对最大耗时做出评估。如果两者相差太大（4 倍），则提示测量结果不可靠。

```
$ python -m timeit  -n 10  -r 3  -s "import time"  "time.sleep(1)"
10 loops, best of 3: 1 sec per loop
```

可在代码中调用测量函数，分别对应算法的前两个步骤。

```
>>> timeit.timeit("time.sleep(1)", setup = "import time", number = 10)
10.037211537011899

>>> timeit.repeat("time.sleep(1)", setup = "import time", number = 10, repeat = 3)
[10.03519637597492, 10.037801038008183, 10.027445557003375]
```

如果创建计时器实例，还可使用其 autorange 方法探测合适的循环次数（number）。这对那些不便手工设定循环数值的测量很有帮助。

```
>>> t = timeit.Timer("time.sleep(0.002)", "import time")
>>> t.autorange()
(100, 0.2538072219467722)
```

默认使用高精度计时器统计 Wall 时间，也可改为统计 CPU 时间。只是返回结果是系统和用户时间的总和，从这一点上来说其不能替代 time 系统命令。

> Wall: time.perf_counter; CPU: time.process_time

```
>>> timeit.timeit("time.sleep(1)", "import time", number = 10, timer = time.process_time)
0.0010270000000001112
```

```
$ python -m timeit -p -n 10 -r 3 -s "import time" "time.sleep(1)"
10 loops, best of 3: 29 usec per loop
```

类似命令可在 IPython、Jupyter Notebook 里使用。

IPython

```
>>> %time time.sleep(1)
CPU times: user 270 µs, sys: 527 µs, total: 797 µs
Wall time: 1 s

>>> %timeit -n 10 -r 3 time.sleep(1)
1 s ± 434 µs per loop (mean ± std. dev. of 3 runs, 10 loops each)

>>> %timeit -c -n 10 -r 3 time.sleep(1)                                  # CPU
56 µs ± 35.1 µs per loop (mean ± std. dev. of 3 runs, 10 loops each)
```

11.2.2 profile

标准库的 profile 模块用于测量执行过程中函数的执行时间和调用次数。

> 标准库自带两个版本，调用接口基本一致。cProfile 以 C 实现，额外开销小，推荐使用。而 profile 是纯 Python 实现，解释器的相关开销会导致测量误差较大，其适合用来编写扩展分析器。

test.py

```
def sum(m):
    n = 0
    for i in range(m):
        n += i
    return n


def main():
    for i in range(3):
        sum(1000000)

    n = 0
    for i in range(100000):
        n += i


if __name__ == '__main__':
    main()
```

```
$ python -m cProfile -s cumtime test.py

         7 function calls in 0.231 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.231    0.231 {built-in method builtins.exec}
        1    0.000    0.000    0.231    0.231 test.py:4(<module>)
        1    0.008    0.008    0.231    0.231 test.py:11(main)
        3    0.224    0.075    0.224    0.075 test.py:4(sum)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler'}
```

---

ncalls：被调用的次数。

tottime：总执行时间，但不包括其调用的子函数。

percall：tottime / ncalls。

cumtime：执行总耗时，包括其调用的子函数。

percall：cumtime / ncalls。

如在代码中使用，建议先将其封装成装饰器或上下文。

decorator

```
import cProfile, pstats, functools, time

def profile(sortby = "cumtime", limit = 10, timer = time.perf_counter):
    def decorator(func):

        @functools.wraps(func)
        def wrap(*args, **kwargs):
            p = cProfile.Profile(timer)
            p.enable()
            try:
                return func(*args, **kwargs)
            finally:
                p.disable()
```

```
                s = pstats.Stats(p).sort_stats(sortby)
                s.print_stats(limit)

        return wrap
    return decorator
```

context

```
import cProfile, pstats, contextlib, time

@contextlib.contextmanager
def profile(sortby = "cumtime", limit = 10, timer = time.perf_counter):
    p = cProfile.Profile(timer)
    p.enable()
    try:
        yield
    finally:
        p.disable()
        s = pstats.Stats(p).sort_stats(sortby)
        s.print_stats(limit)
```

然后，根据需要选用即可。

test.py

```
def sum(m):
    n = 0
    for i in range(m):
        n += i
    return n


@profile()
def main():
    for i in range(3):
        sum(1000000)

    n = 0
    for i in range(100000):
        n += i


if __name__ == '__main__':
```

```
    main()
```

```
$ python test.py

         5 function calls in 0.233 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.007    0.007    0.233    0.233 test.py:46(main)
        3    0.226    0.075    0.226    0.075 test.py:39(sum)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler'}
```

11.2.3 line profiler

获知引发性能问题的函数后，接下来需要进一步分析具体成因。line_profiler 可以对函数内的代码逐行进行测量。

test.py

```
@profile
def test():
    for i in range(3):
        a = [1] * (10 ** 6)
    b = [2] * (2 * 10 ** 7)
    del b
    return a

if __name__ == '__main__':
    test()
```

---

kernprof 创建 LineProfiler 实例，并将 profile 装饰器插入__builtins__，所以在目标代码里无须导入相关模块。这里可为多个执行函数添加装饰器进行测量。

```
$ kernprof -l -v test.py

Timer unit: 1e-06 s
Total time: 0.16758 s
File: test.py
Function: test at line 1
```

```
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           @profile
     2                                           def test():
     3         4           20      5.0      0.0      for i in range(3):
     4         3        18702   6234.0     11.2          a = [1] * (10 ** 6)
     5         1        80352  80352.0     47.9      b = [2] * (2 * 10 ** 7)
     6         1        68504  68504.0     40.9      del b
     7         1            2      2.0      0.0      return a
```

Line：函数内源码的行号。

Hits：执行次数。

Time：总计执行时间。

Per Hit：单次执行的平均时间（time/hits）。

% Time：占总耗时的百分比。

Line Contents：源代码。

只需载入 line_profiler 扩展，便可在 IPython 里使用 lprun 命令。

IPython

```
>>> def test(n):
        a = []
        for i in range(n):
            a.extend([1] * (10 ** 6))
        return a

>>> def run():
        a = test(2)
        return a
```

```
>>> %load_ext line_profiler
```

```
>>> %lprun -f run -f test run()              # -f 指定待测量函数，后面的 run()是启动代码

Timer unit: 1e-06 s

Total time: 0.025681 s
Function: test at line 1
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def test(n):
     2         1            1      1.0      0.0      a = []
     3         3            8      2.7      0.0      for i in range(n):
     4         2        25672  12836.0    100.0          a.extend([1] * (10 ** 6))
     5         1            0      0.0      0.0      return a

Total time: 0.025696 s
Function: run at line 1
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def run():
     2         1        25696  25696.0    100.0      a = test(2)
     3         1            0      0.0      0.0      return a
```

11.2.4 memory profiler

除执行时间外，另一相关测试是内存占用。当程序内存占用过多时，可能会引发物理内存不足，或者导致与硬盘的频繁交换，这些都会引发严重的性能问题。

参考 line_profiler 设计，memory_profiler 用于逐行分析代码内存占用。

> 建议安装 psutil，否则会有 “memory_profiler will be slow” 警告。

test.py

```
@profile
def test():
    a = [1] * (10 ** 6)
    b = [2] * (2 * 10 ** 7)
    del b
    return a
```

```
if __name__ == '__main__':
    test()
```

```
$ python -m memory_profiler test.py

Line #      Mem usage      Increment     Line Contents
================================================
     1     37.625 MiB      0.000 MiB     @profile
     2                                   def test():
     3     45.258 MiB      7.633 MiB         a = [1] * (10 ** 6)
     4    197.848 MiB    152.590 MiB         b = [2] * (2 * 10 ** 7)
     5     45.258 MiB   -152.590 MiB         del b
     6     45.258 MiB      0.000 MiB         return a
```

Mem usage：执行该行之后，解释器内存使用总量。

Increment：相对于上一行的内存使用增量。

同样可在 IPython 内载入扩展，其使用方式与 lprun 类似。

```
>>> %load_ext memory_profiler
>>> %mprun -f test test()
>>> %memit test()
```

除此之外，mprof 命令还可对进程内存进行采样，并使用 matplotlib 输出图形。那些被添加了 profile 装饰器的函数被重点标记。这样就可先从宏观角度观察内存使用趋势。

11.2.5 pympler

与基于代码行的分析相比，pympler 则被用来统计对象实例的内存使用。

summary

首先，我们对整个解释器进程内的存活对象做出评估，以查证哪类对象可能存在问题。

```
>>> from pympler import summary, muppy
```

```
>>> all_objects = muppy.get_objects()
>>> len(all_objects)
144074

>>> sum1 = summary.summarize(all_objects)
>>> summary.print_(sum1)

                types |   # objects |   total size
==================== | =========== | ============
         <class 'str |       55374 |      6.14 MB
        <class 'list |       12696 |      3.30 MB
        <class 'dict |        9507 |      3.25 MB
        <class 'code |       10632 |      1.47 MB
        <class 'type |        1374 |      1.40 MB
         <class 'set |        1778 |    707.69 KB
       <class 'tuple |       10386 |    672.27 KB
     <class 'weakref |        3318 |    259.22 KB
         <class 'int |        4764 |    141.39 KB
                               ...
```

tracker

接下来，在不同执行点创建跟踪快照。这样通过输出快照的差异，就可能发现嫌疑目标，以及引发问题的代码位置。

```
>>> from pympler.tracker import SummaryTracker
>>> tr = SummaryTracker()

>>> x = 100
>>> tr.print_diff()

            types |   # objects |   total size
================= | =========== | ============
     <class 'list |        6663 |    625.98 KB
      <class 'str |        6670 |    481.73 KB
                              ...
```

```
>>> s = "abc"
>>> tr.print_diff()

            types |   # objects |   total size
```

```
================ | ========== | ============
      <class 'str |          9 |    627     B
     <class 'list |          4 |    184     B
                            ...
```

可进一步缩小范围，只跟踪某些具体类型。

```
>>> from pympler.classtracker import ClassTracker

>>> class A: pass
>>> class B: pass

>>> tr = ClassTracker()
>>> tr.track_class(A)
>>> tr.track_class(B)
>>> tr.create_snapshot()
```

```
>>> a = A()
>>> b = A()
>>> c = B()

>>> tr.create_snapshot()
>>> tr.stats.print_summary()              # 输出两次快照对象的数量和内存占用对比

---- SUMMARY ------------------------------------------------------------
                                       active      0     B      average   pct
  __main__.A                                0      0     B      0     B    0%
  __main__.B                                0      0     B      0     B    0%
                                       active      0     B      average   pct
  __main__.A                                2    336     B    168     B    0%
  __main__.B                                1    168     B    168     B    0%
-------------------------------------------------------------------------
```

```
>>> d = B()
>>> del a

>>> tr.create_snapshot()
>>> tr.stats.print_summary()              # 输出 3 次快照对比

---- SUMMARY ------------------------------------------------------------
                                       active      0     B      average   pct
  __main__.A                                0      0     B      0     B    0%
```

```
  __main__.B                                0      0     B      0     B    0%
                                        active      0     B      average   pct
  __main__.A                                2    336     B    168     B    0%
  __main__.B                                1    168     B    168     B    0%
                                        active      0     B      average   pct
  __main__.A                                1    184     B    184     B    0%
  __main__.B                                2    336     B    168     B    0%
-------------------------------------------------------------------------------
```

sizeof

找到嫌疑目标后，我们开始对其进行个体检查，这需要使用 asizeof。与 sys.getsizeof 不同，它递归统计所有成员和属性。

> 对于引用，sys.getsizeof 仅计入一个指针大小，而对所引用对象的变动漠不关心。而 pympler.asizeof 则因递归统计，能获知所有引用目标的内存变化。

```
>>> from sys import getsizeof
>>> from pympler.asizeof import asizeof

>>> a = []
>>> b = [a]

>>> getsizeof(b), asizeof(b)
(72, 136)
```

```
>>> a.append(1)                          # b 所引用的 a 列表发生变化
>>> getsizeof(b), asizeof(b)             # asizeof 能“感知”到大小变化
(72, 200)
```

递归统计同样适用于自定义类型的属性变化。

```
>>> class A: pass
>>> a = A()

>>> getsizeof(a), asizeof(a)
(56, 168)
```

```
>>> a.x = 100
```

```
>>> getsizeof(a), asizeof(a)
(56, 256)
```

为便于排查，还可输出详细统计信息。

```
>>> from pympler.asizeof import asized

>>> o = [1, [1, "abc", (100, 200)]]

>>> print(asized(o, detail = 1).format())
[1, [1, 'abc', (100, 200)]] size=384 flat=80
    [1, 'abc', (100, 200)]  size=272 flat=88
    1                       size=32  flat=32
```

```
>>> class A: pass

>>> a = A()
>>> a.x  = 100
>>> a.y = "hello, world!"
>>> a.z = [1, 2, 3]

>>> print(asized(a, detail = 2).format())
<A object at 0x106e35cf8> size=616 flat=56
    __dict__ size=560 flat=112
        [K] x size=56 flat=56
        [K] y size=56 flat=56
        [K] z size=56 flat=56
        [V] x: 100             size=32  flat=32
        [V] y: 'hello, world!' size=64  flat=64
        [V] z: [1, 2, 3]       size=184 flat=88
   __class__ size=0 flat=0
```

第 12 章　工具

12.1　调试器

与在代码中使用固定测试语句相比，调试器显然更加灵活，其适合完成一些随机操作。

- 单步执行代码。
- 按条件设置断点。
- 查看执行栈帧信息。
- 在特定上下文执行代码。
- 回溯异常现场，查看问题所在。

标准库自带 PDB 调试器，其操作方式与 GDB 类似。

我们可使用更完善的 iPDB 来代替它，iPDB 支持语法高亮，且输出更加友好。

初学者还可选用 PuDB，这是一种类似于 Turbo C 界面的调试器。

启动

在命令行下以调试方式启动程序。

```
$ python -m pdb main.py
```

还可在源码的特定位置插入断点，启动调试器。

```
import pdb
pdb.set_trace()
```

或者直接以调试方式执行代码。

```
pdb.run("test()")
```

> 另有 runcall、runeval 用于函数和表达式。

另有一种被称作 Post-Mortem 的模式。当异常发生时，其允许我们像法医那样到事故现场进行验尸，以查证相关问题。

> 使用 pdb 启动程序，在异常发生时，会自动进入“验尸”模式。其他环境，则可调用 pdb.post_mortem 方法进入。至于 IPython，直接使用内置%pdb 开关即可。
>
> 因为异常发生时，调用堆栈帧被 traceback 保存，所以很容易恢复现场，获取状态数据。

```
def main():
    try:
        test()
    except:
        pdb.post_mortem()
```

更简洁的做法是以-i 参数启动解释器。

```
$ python -i main.py

Traceback (most recent call last):
  File "main.py", line 17, in <module>
    main()
  File "main.py", line 13, in main
    test()
  File "main.py", line 8, in test
```

```
    raise Exception
Exception

>>> import pdb
>>> pdb.pm()
> main.py(8)test()
-> raise Exception
(pdb)
```

命令

命令不算多，稍加练习便可适应。

```
(pdb) help

Documented commands (type help <topic>):
========================================
alias,unalias  continue|c                ignore           p|pp             step|s
args|a         debug                     interace         quit|q,exit      tbreak
break|b        disable,enable            jump|j           restart,run      until|unt
clear|cl       display,undisplay         list|l           return|r         whatis
commands       down|d,up|u               longlist|ll      retval|rv        where|w,bt
condition      help|h                    next|n           source

Miscellaneous help topics:
==========================
exec  pdb
```

```
(pdb) help a
a(rgs): Print the argument list of the current function.
```

断点

基于文件名、行号或函数设置断点。

```
(pdb) b demo.py:3                              # 指定文件名和行号
Breakpoint 1 at demo.py:3

(pdb) b 7                                      # 省略当前文件名，仅指定行号
Breakpoint 2 at main.py:7
```

```
(pdb) b demo.test                                    # 函数全名
Breakpoint 3 at demo.py:1

(pdb) b main                                         # 当前模块内的函数
Breakpoint 4 at main.py:6
```

省略参数时，会列出全部断点。

```
(pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at demo.py:3
2   breakpoint   keep yes   at main.py:7
3   breakpoint   keep yes   at demo.py:1
4   breakpoint   keep yes   at main.py:6
```

可禁用或重新启用某些断点。

```
(pdb) disable 2 4                                    # 指定一到多个断点序号
Disabled breakpoint 2 at main.py:7
Disabled breakpoint 4 at main.py:6

(pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at demo.py:3
2   breakpoint   keep no    at main.py:7
3   breakpoint   keep yes   at demo.py:1
4   breakpoint   keep no    at main.py:6
```

```
(pdb) enable 2 4
Enabled breakpoint 2 at main.py:7
Enabled breakpoint 4 at main.py:6

(pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at demo.py:3
2   breakpoint   keep yes   at main.py:7
3   breakpoint   keep yes   at demo.py:1
4   breakpoint   keep yes   at main.py:6
```

执行循环时，可能需要暂时忽略某个断点。

如省略次数，或计数器减归零，则此忽略设置被清除，断点恢复。

```
(pdb) ignore 1 3
Will ignore next 3 crossings of breakpoint 1.

(pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at demo.py:3
        ignore next 3 hits
2   breakpoint   keep yes   at main.py:7
3   breakpoint   keep yes   at demo.py:1
4   breakpoint   keep yes   at main.py:6
```

为断点设定条件，也能避免执行被频繁中断。

```
(pdb) condition 2 n > 5
New condition set for breakpoint 2.

(pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at demo.py:3
2   breakpoint   keep yes   at main.py:7
        stop only if n > 5
3   breakpoint   keep yes   at demo.py:1
4   breakpoint   keep yes   at main.py:6
```

避免重复操作，可提前设定断点触发时要执行的调试命令。

重新设置，并以 end 直接结束来取消自动命令。

```
(pdb) commands 1
(com) args
(com) p n
(com) end
```

最后，删除某些或全部断点。

```
(pdb) cl 1 2
```

```
Deleted breakpoint 1 at demo.py:3
Deleted breakpoint 2 at at main.py:7

(pdb) cl
Clear all breaks? y
```

也可以 tbreak 设置临时断点，在命中后自动删除。

```
(pdb) tbreak demo.test
Breakpoint 5 demo.py:1

(pdb) b
Num Type         Disp Enb   Where
5   breakpoint   del  yes   at demo.py:1
```

执行

执行程序，直到预设断点处暂停。查看上下文及对象状态，执行相关命令。然后单步执行，或设定新断点继续。

命令 continue 表示执行到下一断点，next 表示在当前函数单步执行，而 step 则会进入可调用对象内。

---

main.py

```
 1  def test(n):
 2      x = 0
 3      for i in range(n):
 4          x += i
 5          print(i, x)
 6      return x
 7
 8
 9  def main():
10      print(test(5))
11      print("end.")
12
13
14  if __name__ == '__main__':
```

```
15      main()
```

```
(pdb) b 10
Breakpoint 1 at main.py:10

(pdb) c                                    # 执行到断点处停止
> main.py(10)main()
-> print(test(5))

(pdb) n                                    # 仅在当前函数单步执行，不会进入 test 函数
0 0
1 1
2 3
3 6
4 10
10
> main.py(11)main()
-> print("end.")

(pdb) n
end.
```

```
(pdb) run                                  # 重新启动，单保留上次断点等设置
Restarting main.py

(pdb) c                                    # 执行到断点
> test/main.py(10)main()
-> print(test(5))

(pdb) s                                    # 进入 test 函数内部
--Call--
> main.py(1)test()
-> def test(n):
```

> 在提示符后直接回车，则继续执行上一次的命令，避免重复输入相同的命令。

要跳出循环或函数，须使用另外两个命令。

until 执行到行号大于当前的位置，这意味着你得先执行到循环尾部。

return 直接执行到函数的返回位置，但要避免断点干扰。

```
(pdb) tbreak 5
Breakpoint 1 at main.py:5

(pdb) c
Deleted breakpoint 1 at main.py:5
> main.py(5)test()
-> print(i, x)

(pdb) unt
0 0
1 1
2 3
3 6
4 10
> main.py(6)test()
-> return x
```

```
(pdb) b test
Breakpoint 1 at main.py:1

(pdb) c
> main.py(2)test()
-> x = 0

(pdb) r
0 0
1 1
2 3
3 6
4 10
--Return--
> main.py(6)test()->10
-> return x
```

还可在当前作用域内，按行号无条件跳转。比如回到函数的起始处重新执行；或者忽略中间代码，直接到退出位置。注意，这与断点不同。

```
(pdb) tbreak 5
Breakpoint 1 at main.py:5

(pdb) c
Deleted breakpoint 1 at main.py:5
> main.py(5)test()
-> print(i, x)

(pdb) l
  1     def test(n):
  2         x = 0
  3         for i in range(n):
  4             x += i
  5  ->         print(i, x)
  6         return x

(pdb) j 2                                   # 回到函数的起始处，重新执行。但不会重置上下文状态
> main.py(2)test()
-> x = 0

(pdb) n
> main.py(3)test()
-> for i in range(n):

(pdb) j 6                                   # 跳转到函数的退出位置，中间被忽略的代码未曾执行
> main.py(6)test()
-> return x

(pdb) n
--Return--
> main.py(6)test()->0
-> return x
```

堆栈

在调试过程中，我们应随时掌握调用状况，知道执行过程、当前位置，以及堆栈帧的状态，并以此为上下文，输出或修改相关对象值。

> 命令 where 输出整个调用堆栈，而 up、down 则选择要操作的栈帧。

```
(pdb) b test
Breakpoint 1 at main.py:1

(pdb) c
> main.py(2)test()
-> x = 0

(pdb) w
  main.py(15)<module>()
-> main()
  --------------------
  main.py(10)main()
-> print(test(5))
  --------------------
> main.py(2)test()                    # 当前栈帧（注意前面的右尖括号提示）
-> x = 0
```

从输出效果来看，ipdb 要远远好于 pdb。

```
(pdb) u                               # 选择上（前）一栈帧
> main.py(10)main()
-> print(test(5))

(pdb) w
  main.py(15)<module>()
-> main()
--------------------
> main.py(10)main()
-> print(test(5))
--------------------
  main.py(2)test()
-> x = 0
```

```
(pdb) d                               # 回到下一栈帧
> main.py(2)test()
-> x = 0

(pdb) w
  main.py(15)<module>()
-> main()
--------------------
```

```
  main.py(10)main()
-> print(test(5))
------------------
> main.py(2)test()
-> x = 0
```

查看

比如查看当前源码，以确定接下来要做什么。

---

list 默认输出 11 行代码，可指定起始和结束行号。箭头指明当前执行位置。

longlist 仅输出当前函数（栈帧）的全部源码。

source 输出某个对象（模块或函数）的源码。

```
(pdb) l
  1 B   def test(n):
  2  ->     x = 0
  3         for i in range(n):
  4             x += i
  5             print(i, x)
  6         return x
  7
  8
  9     def main():
 10         print(test(5))
 11         print("end.")
```

---

args 输出函数调用参数，p/pp 输出对象或表达式结果，whatis 查看对象类型。

retval 查看当前函数的返回值（__return__）。

```
(pdb) args
n = 5

(pdb) p hex(n)
'0x5'
```

```
(pdb) whatis n
<class 'int'>
```

这里还可以自动监视某个对象值是否改变。

```
(pdb) l
  1  -> def test(n):
  2         x = 0
  3         for i in range(n):
  4             x += i
  5             print(i, x)
  6         return x

(pdb) b 5
Breakpoint 1 at main.py:5

(pdb) c
> main.py(5)test()
-> print(i, x)

(pdb) display x                          # 监视 x。当值改变时，输出相关信息
display x: 0

(pdb) c
0 0
> main.py(5)test()
-> print(i, x)
display x: 1  [old: 0]

(pdb)
1 1
> main.py(5)test()
-> print(i, x)
display x: 3  [old: 1]
```

其在无参数时输出所有设置，或用 undisplay 取消。

```
(pdb) display
Currently displaying:
x: 3
```

```
(pdb) undisplay x
```

其他

可直接执行 Python 语句。但为避免和调试命令冲突，建议添加感叹号前缀。

> 感叹号后不能有空格，否则提示缩进错误。

```
(pdb) !c = 100

(pdb) p c
100
```

如果要执行的命令或语句太长，则可为其创建别名。

```
(pdb) alias c_globals pp {k:v for k, v in globals().items() if "%1" not in k}
(pdb) c_globals __
{'main': <function main at 0x102a39ea0>, 'test': <function test at 0x102a39f28>}
```

```
(pdb) alias
c_dir = pp [n for n in dir() if "%1" not in n]
c_globals = pp {k:v for k, v in globals().items() if "%1" not in k}

(pdb) unalias c_globals
```

保留当前调试状态，临时去做点别的什么也是允许的。

> interact 将当前的 globals、locals 名字空间合并，启动一个 Python Shell。
>
> debug 则是为指定代码（比如函数调用）启动一个新的调试器。

```
(pdb) l
  1 B   def test(n):
  2  ->     x = 0
  3         for i in range(n):
  4             x += i
  5             print(i, x)
```

```
  6         return x

(pdb) debug test(10)
ENTERING RECURSIVE DEBUGGER

((pdb)) s
--Call--
> main.py(1)test()
-> def test(n):

((pdb)) n
> main.py(2)test()
-> x = 0

((pdb))
> main.py(3)test()
-> for i in range(n):

((pdb)) exit
LEAVING RECURSIVE DEBUGGER
```

调试告一段落，之后要么退出，要么重启。

---

run：重启，可重新指定启动参数。断点等设置会被保留。

exit、quit：终止程序，退出调试器。

```
$ python -m pdb main.py 1 2

> main.py(1)<module>()
-> def test(n):

(pdb) !import sys
(pdb) !sys.argv
['main.py', '1', '2']

(pdb) b main
Breakpoint 1 at main.py:9

(pdb) b test
Breakpoint 2 at main.py:1
```

```
(pdb) run a b
Restarting main.py with arguments:
        main.py 1 2
> main.py(1)<module>()
-> def test(n):

(pdb) !import sys
(pdb) !sys.argv
['main.py', 'a', 'b']

(pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at main.py:9
2   breakpoint   keep yes   at main.py:1
```

12.2 包管理

PyPA 推荐使用 PIP 作为包管理工具。

> 与以前的 Easy_install + Egg 组合相比，现在使用 PIP + Wheel。Egg 格式于 2004 年启用；而 Wheel 则是 2012 年的全新替代品，是当前 Python 内置标准（PEP 427）。
>
> 在 http://pythonwheels.com/网站中，可以看到主流项目都已使用 Wheel 格式。
>
> 而且，最新版 Python（2.7.9 以上，或者 3.4 以上）都已自带 PIP。

可从网站下载其最新版安装，或使用更新命令。

```
$ wget https://bootstrap.pypa.io/get-pip.py
$ python get-pip.py
```

```
$ pip install -U pip
```

即便在同一系统里，也可为 Python 2 和 Python 3 分别安装 PIP，且支持 VirtualEnv 等虚拟环境。安装后，查看版本及安装路径。

```
$ pip2 -V
pip 9.0.1 from /usr/local/lib/python2.7/site-packages (python 2.7)

$ pip3 -V
pip 9.0.1 from /usr/local/lib/python3.6/site-packages (python 3.6)
```

使用

其命令参数比较多，可使用 help 查看具体帮助信息。

```
$ pip help install
```

install

默认搜索 PyPI，下载并安装最新版本，也可指定版本号。

```
$ pip install SomePackage               # 最新版本
$ pip install SomePackage==1.0.4        # 指定版本
$ pip install 'SomePackage>=1.0.4'      # 指定最小版本
```

```
$ pip install -U requests

Collecting requests
  Downloading requests-2.18.4-py2.py3-none-any.whl (88kB)
    100% |████████████████████████████████| 92kB 169kB/s
...
Successfully installed requests-2.18.4
```

- -e：本地路径，或 Git 等版本控制系统路径。
- -r：基于 requirement file 安装。
- -U：更新包。
- --no-cache-dir：禁用本地缓存，始终从 PyPI 下载。

如果仅下载不安装，可使用 download 命令。

---

uninstall

删除已安装包。

```
$ pip uninstall -y requests

Uninstalling requests-2.18.4:
  Successfully uninstalled requests-2.18.4
```

- -r：删除 requirement file 内列出的所有包。
- -y：yes。

list

列出已安装包。

```
$ pip list -o

Package    Version Latest Type
---------- ------- ------ -----
matplotlib 2.0.2   2.1.0  wheel
```

- -o：仅列出版本过时的包。
- -u：仅列出已是最新版本的包。
- -l：仅列出在虚拟环境下安装的包。

show

显示已安装包的信息。

```
$ pip show requests

Version: 2.18.4
Summary: Python HTTP for Humans.
Home-page: http://python-requests.org
Requires: chardet, certifi, urllib3, idna
```

- -f：显示安装文件列表。

search

从 PyPI 搜索包，支持通配符。

```
$ pip search pympler

Pympler (0.5)  - measure, monitor and analyze the memory behavior of Python objects.
  INSTALLED: 0.5 (latest)
```

requirements files

将当前环境的所有包名称导出到文件。

```
$ pip freeze > requirements.txt
```

文本格式如下，这里每行有一个包名，可含版本号。

requirements.txt

```
pkg1
pkg2==5.3.1
pkg3>=1.0,<=2.0
```

> 高楼目尽欲黄昏，梧桐叶上萧萧雨。——晏殊
>
> 截稿时正赶上一场秋雨，稍有些冷。关窗时莫名想起这句，引之以作结语。

上卷完